P&S Heterogeneous Systems

Advanced Tiling for Matrix Multiplication

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich

Fall 2022

28 November 2022

Parallel Patterns

Reduction Operation

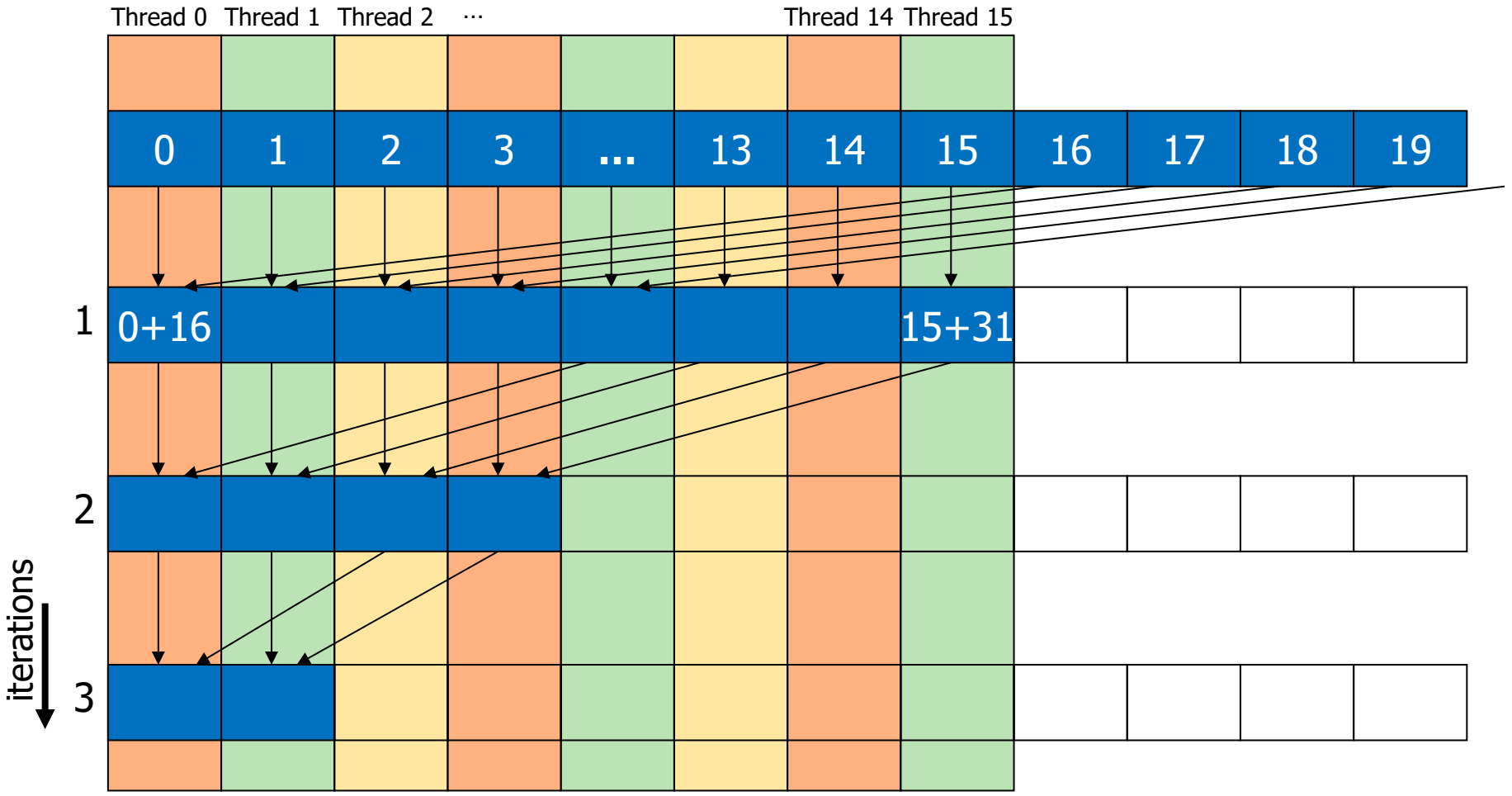
- A reduction operation reduces a set of values to a single value
  - Sum, Product, Minimum, Maximum are examples

- Properties of reduction
  - Associativity
  - Commutativity
  - Identity value

- Reduction is a key primitive for parallel computing
  - E.g., MapReduce programming model

Divergence-Free Mapping (I)

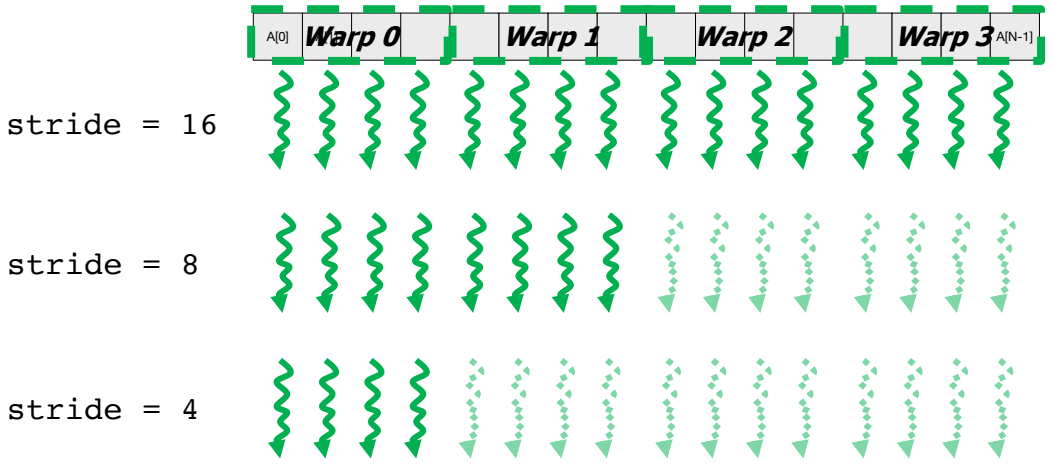
- All active threads belong to the same warp



Divergence-Free Mapping (II)

- Program with high SIMD utilization

```
__shared__ float partialSum[]

unsigned int t = threadIdx.x;

for(int stride = blockDim.x; stride > 0;  stride >> 1){

  __syncthreads();

  if (t < stride)
    partialSum[t] += partialSum[t + stride];

}
```



Warp utilization is maximized

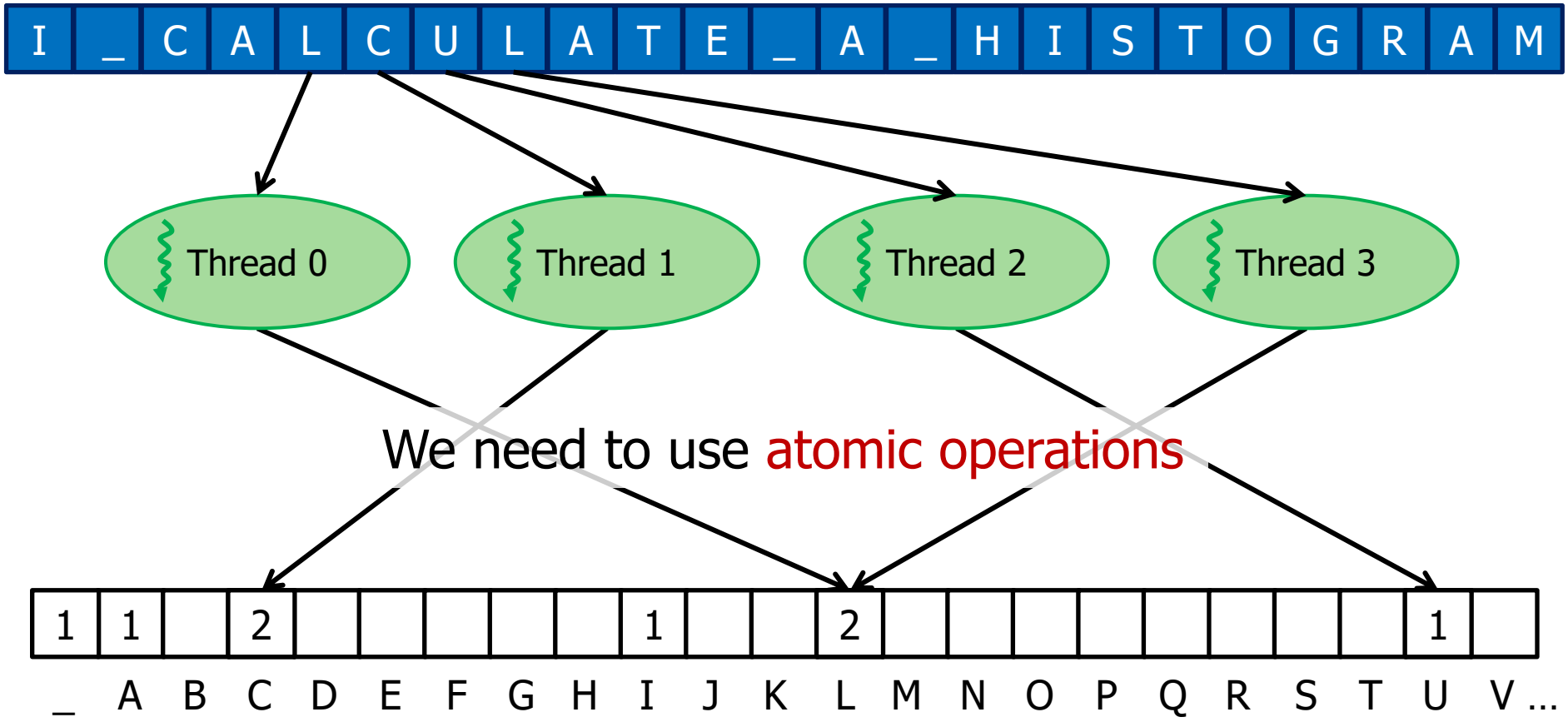
Histogram Computation

- Histogram is a frequently used computation for reducing the dimensionality and extracting notable features and patterns from large data sets
  - Feature extraction for object recognition in images
  - Fraud detection in credit card transactions
  - Correlating heavenly object movements in astrophysics
  - …
- Basic histograms - for each element in the data set, use the value to identify a "bin" to increment
  - Divide possible input value range into "bins"
  - Associate a counter to each bin
  - For each input element, examine its value and determine the bin it falls into and increment the counter for that bin

Slide credit: Hwu & Kirk

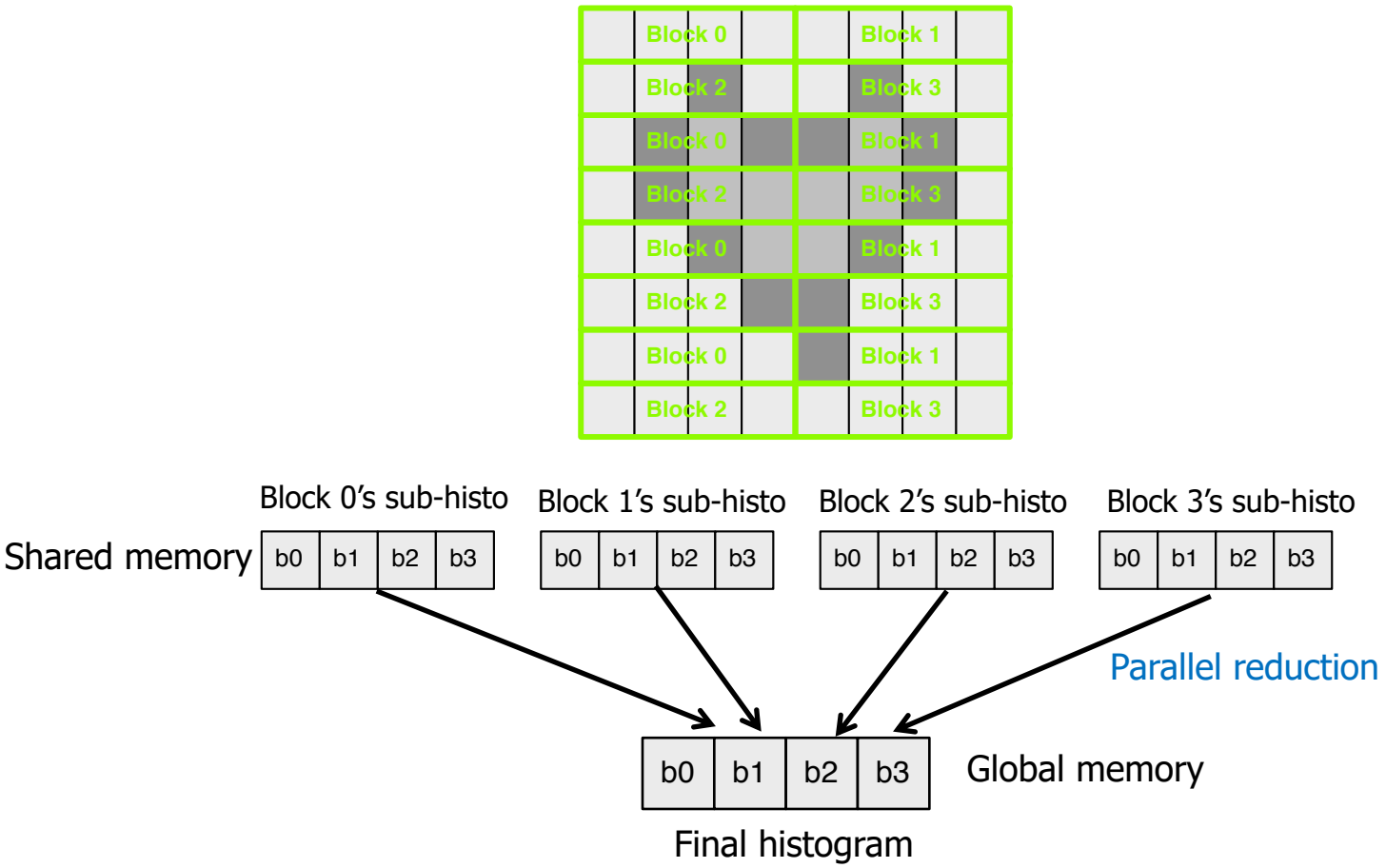
Parallel Histogram Computation: Iteration 2

- All threads move to the next section of the input
  - Each thread moves to element threadID + #threads



Histogram Privatization

- Privatization: Per-block sub-histograms in shared memory
  - Threads use atomic operations in shared memory



Convolution Applications

- Convolution is a widely-used operation in signal processing, image processing, video processing, and computer vision

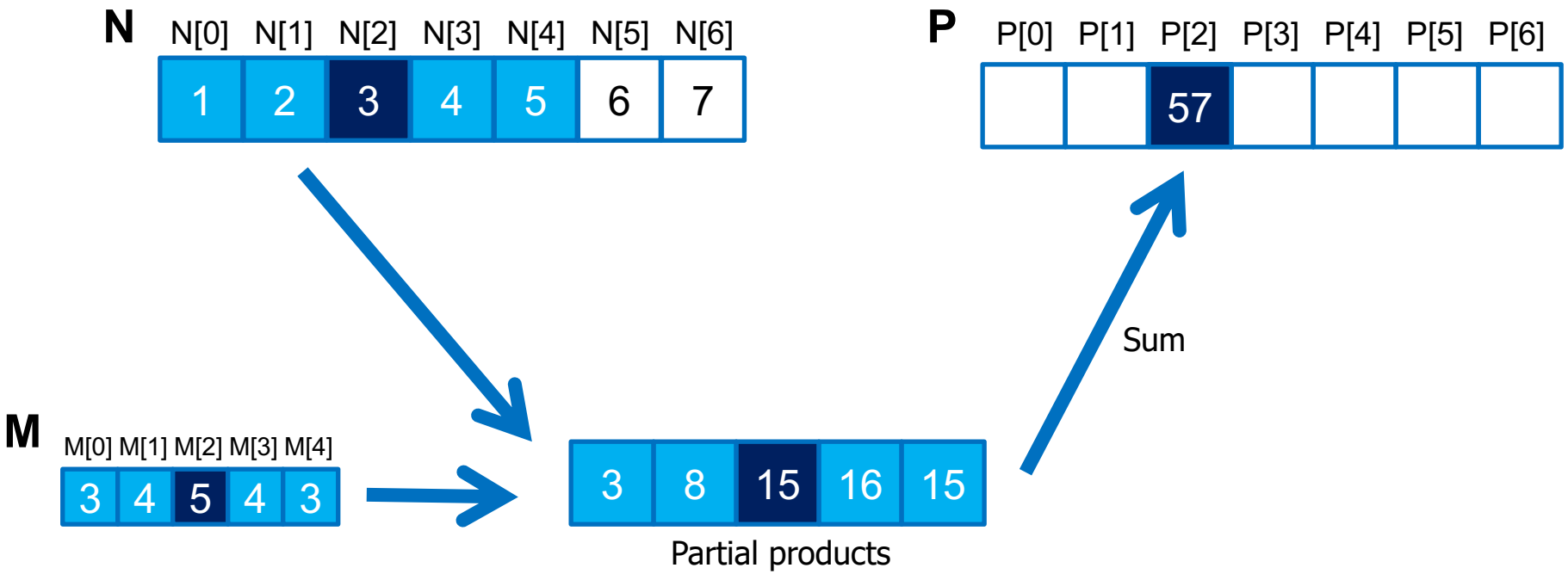
- Convolution applies a filter or mask or kernel* on each element of the input (e.g., a signal, an image, a frame) to obtain a new value, which is a weighted sum of a set of neighboring input elements
  - Smoothing, sharpening, or blurring an image
  - Finding edges in an image
  - Removing noise, etc.

- Applications in machine learning and artificial intelligence
  - Convolutional Neural Networks (CNN or ConvNets)

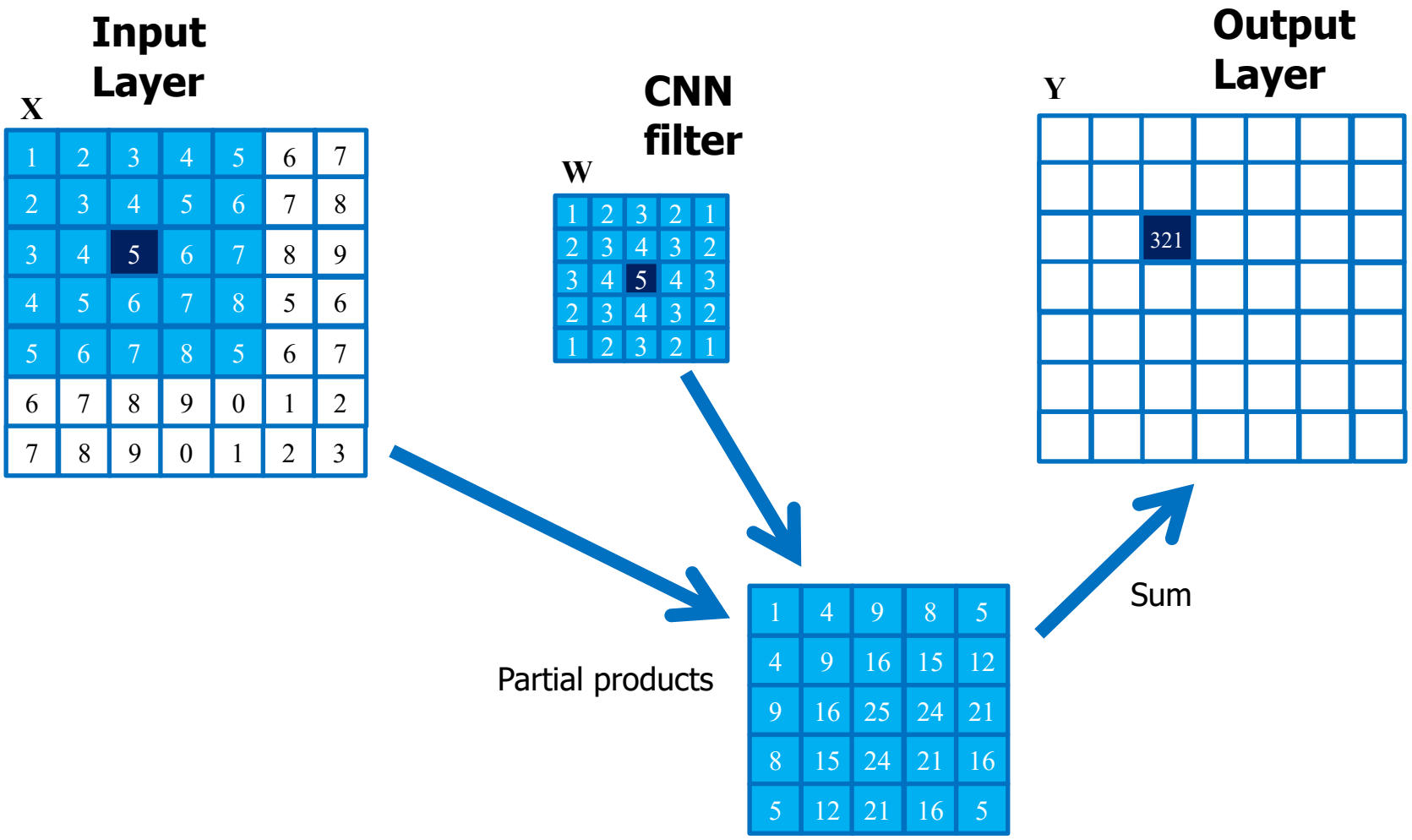
* The term "kernel" may create confusion in the context of GPUs (recall a CUDA/GPU kernel is a function executed by a GPU)

1D Convolution Example

- Commonly used for audio processing
- Mask size is usually an odd number of elements for symmetry (5 in this example)
- Calculation of P[2]:

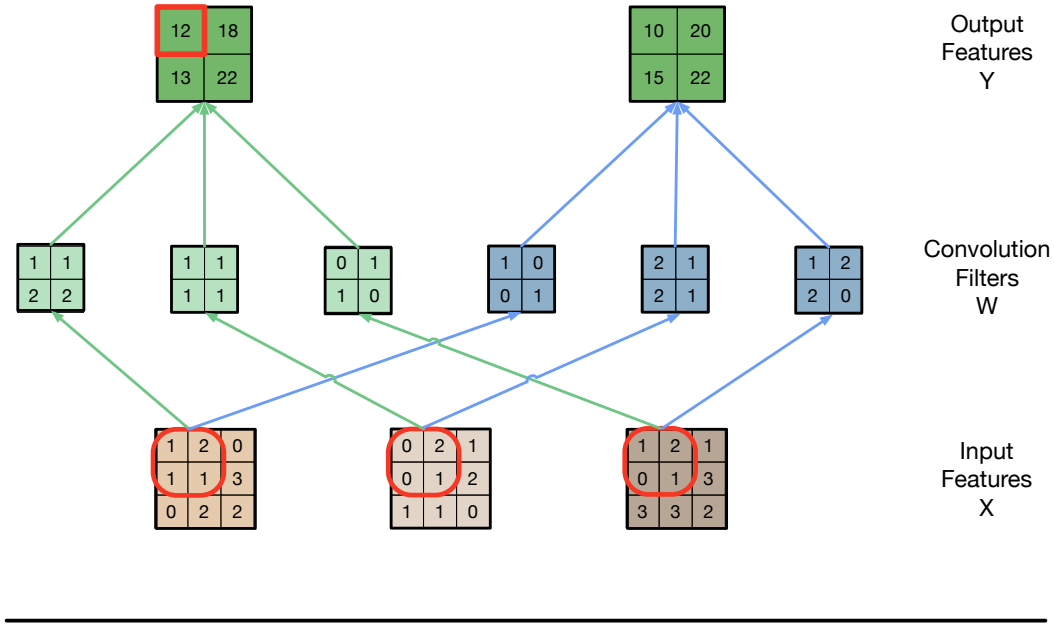


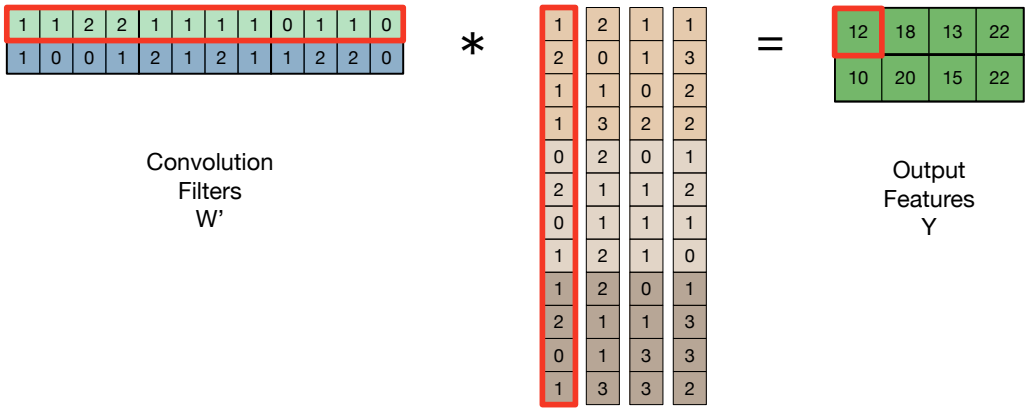
Another Example of 2D Convolution



Slide credit: Hwu & Kirk

Implementing a Convolutional Layer with Matrix Multiplication



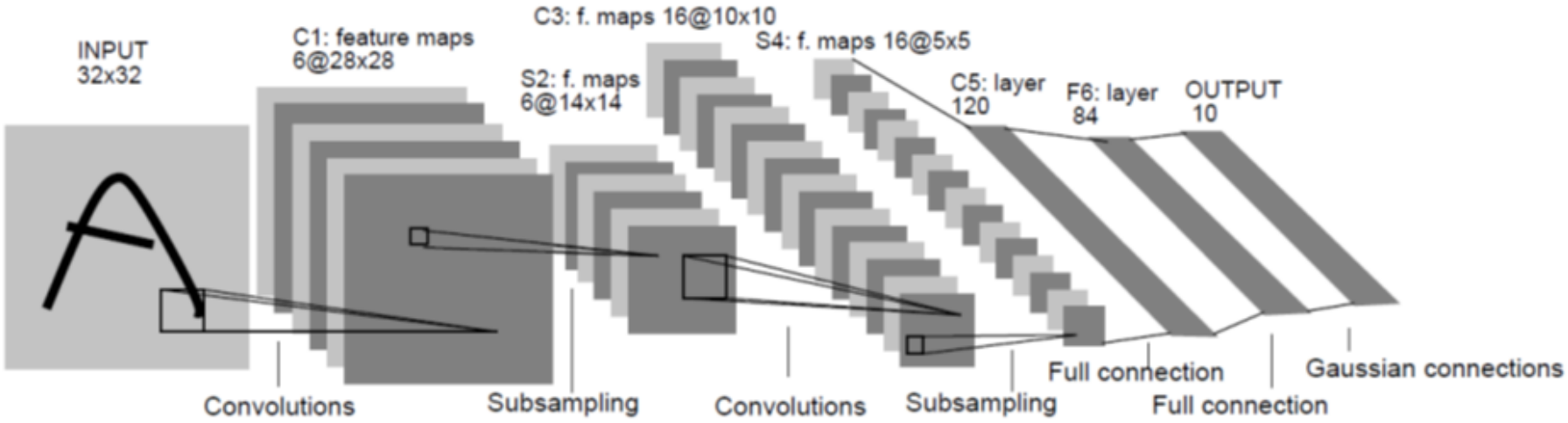


Slide credit: Reproduced from Hwu & Kirk

Efficient CNN on GPU

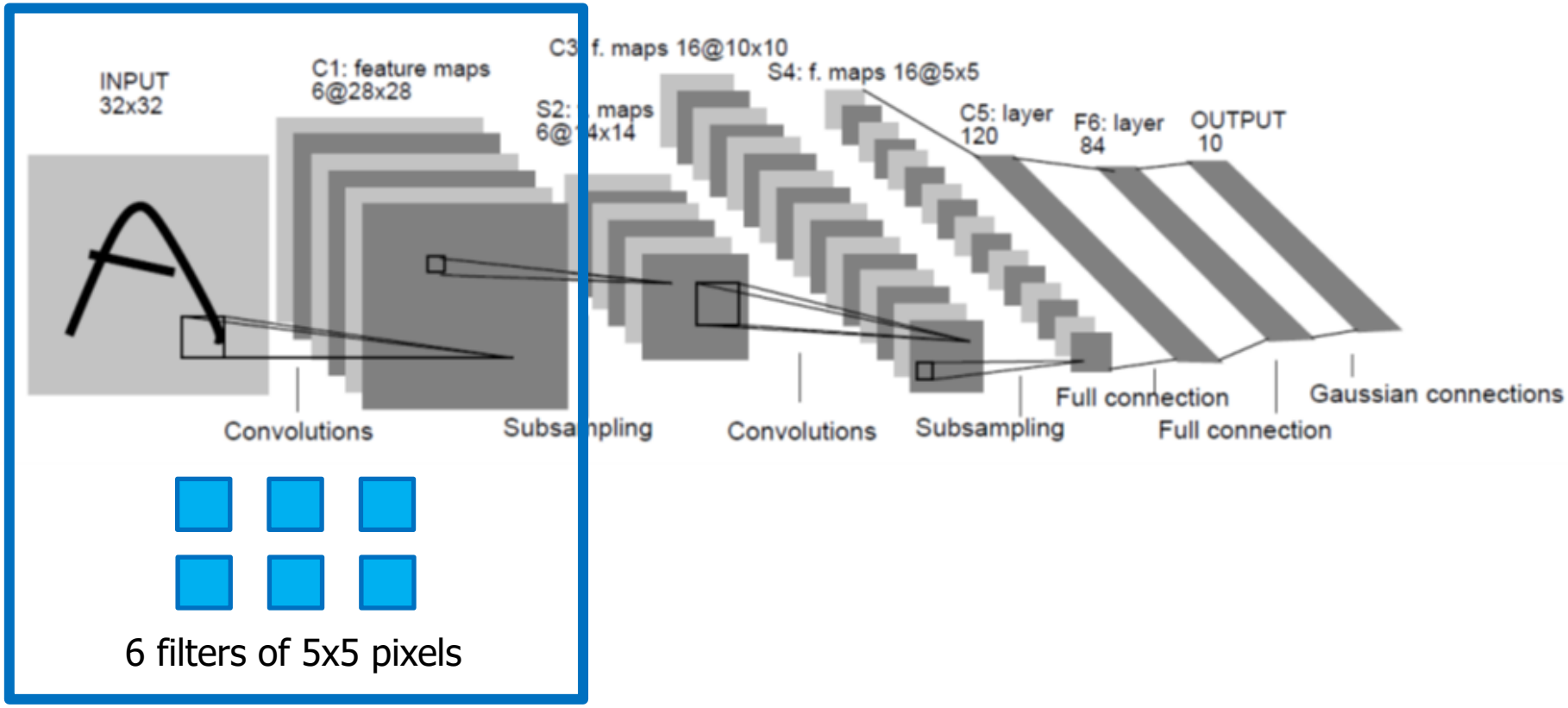
Convolutional Neural Networks

- Neural network with Convolutional layers (combined with other types)
- Example: LeNet-5, CNN for Hand-Written Digit Recognition

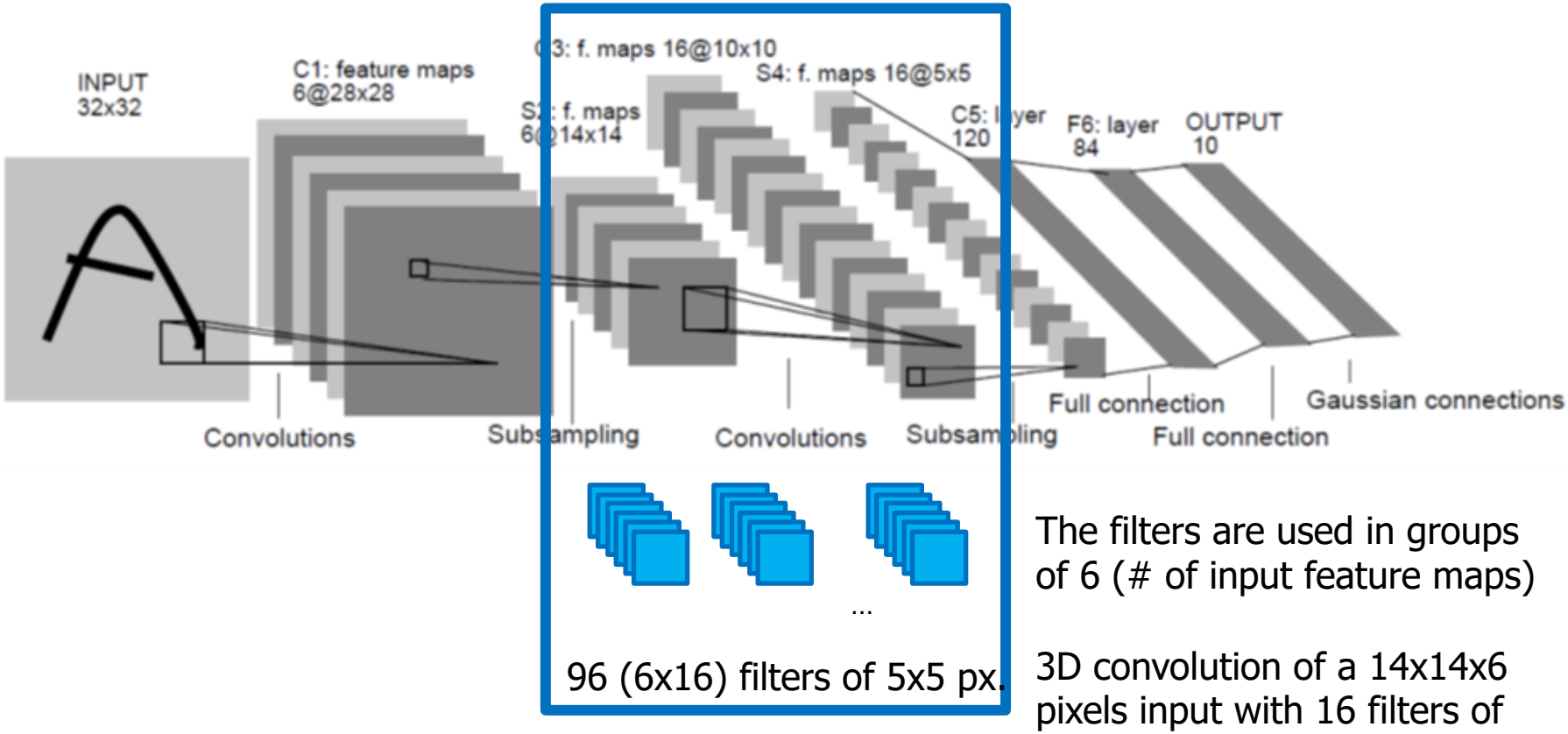


Slide credit: Hwu & Kirk & PUMPS+AI 2022

LeNet-5: CNN for Hand-Written Digit Recognition



LeNet-5: CNN for Hand-Written Digit Recognition

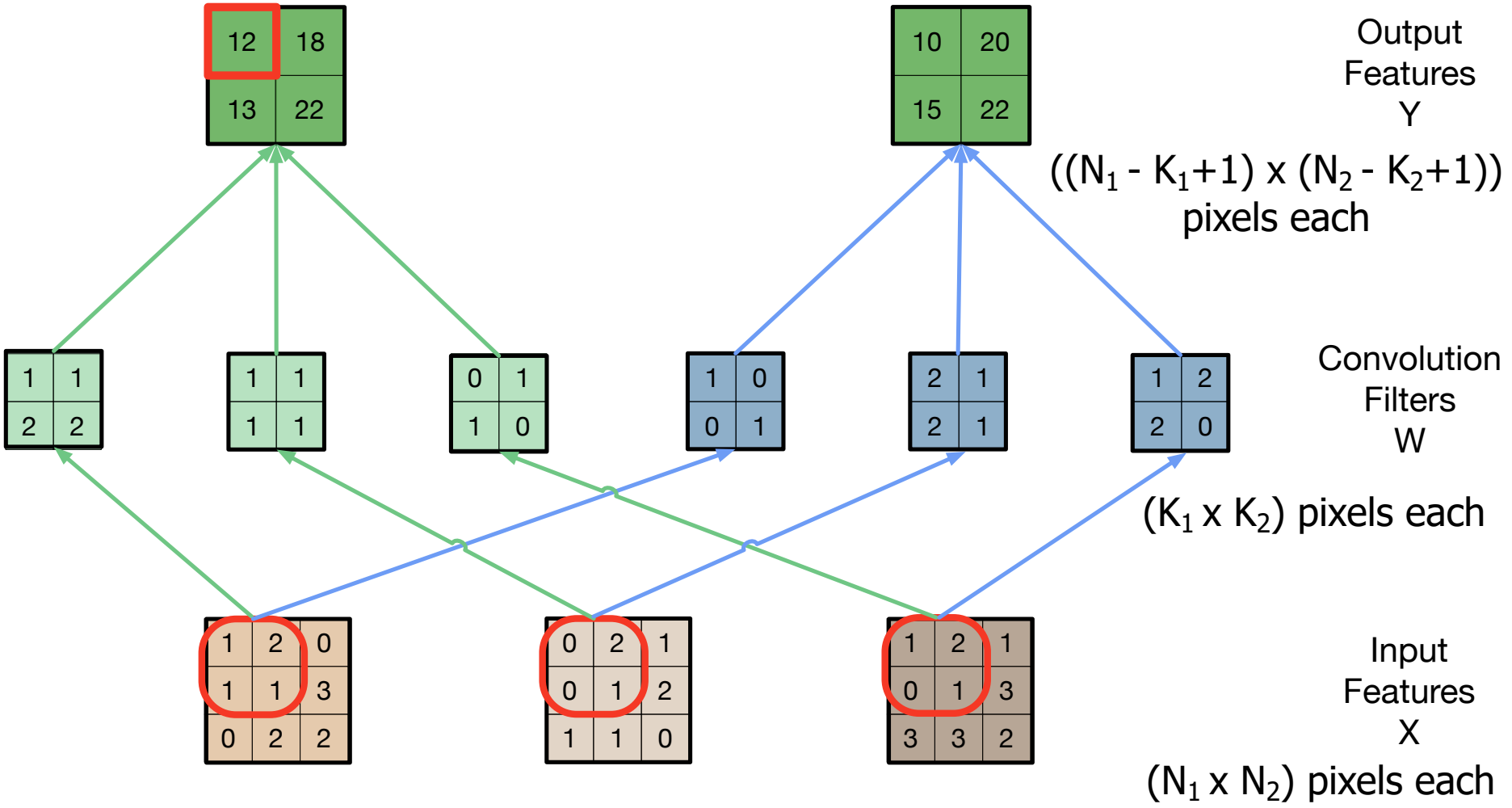


The filters are used in groups of 6 (# of input feature maps)

3D convolution of a 14x14x6 pixels input with 16 filters of 5x5x6 pixels each.

Slide credit: Hwu & Kirk & PUMPS+AI 2022

Example of the Forward Path of a Convolutional Layer



Parallelism in a Convolution Layer

- All output feature map pixels can be calculated in parallel
  - Launch a single kernel to calculate all pixels of all output feature maps
  - Each thread block calculates a tile of output feature map pixels
  - Essentially a collection of summation of 2D tiled convolutions

- See Kirk, Hwu, and El Hajj, PMPP 4th Edition Chapter 16 for such a kernel design

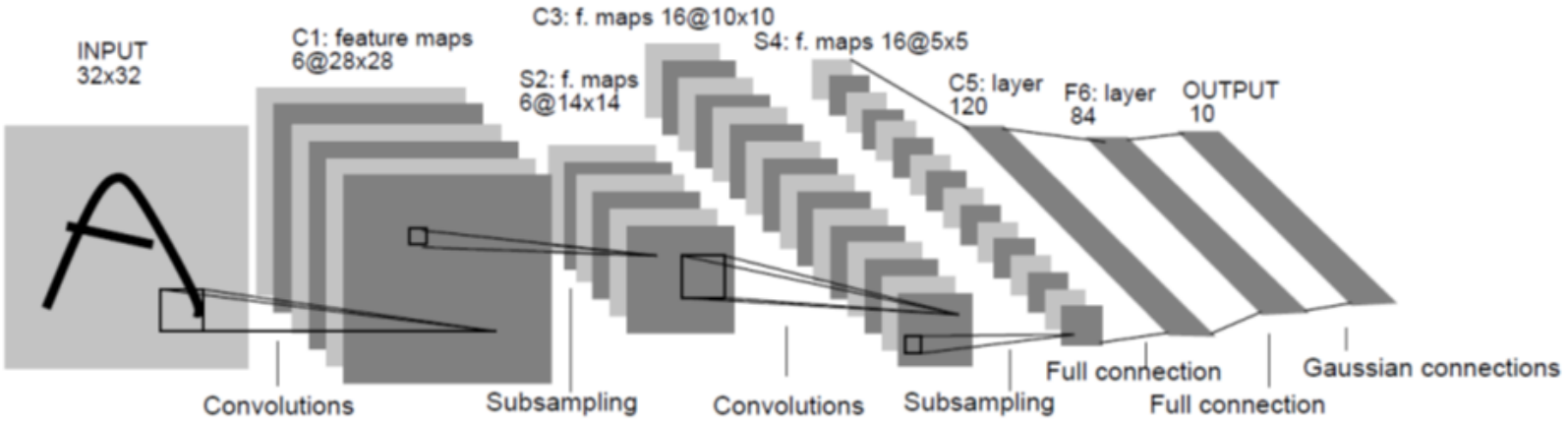
One possible Design of a Basic Kernel

- Each block computes a tile of output pixels
  - TILE_WIDTH pixels in each dimension

- Thread Blocks Grid
  - The X dimension maps to the M output feature maps
  - The Y dimension maps to the tiles in the output feature maps (linearization)

Some Observations

- The amount of parallelism is quite high as long as the total number of pixels across all output feature maps is large
  - For early stages, there are fewer but larger output feature maps (large Y dimension)
  - For later stages, there are more but smaller output feature maps (large X dimension)
  - This scheme keeps the total number of threads stable across the stages



Slide credit: Hwu & Kirk & PUMPS+AI 2022

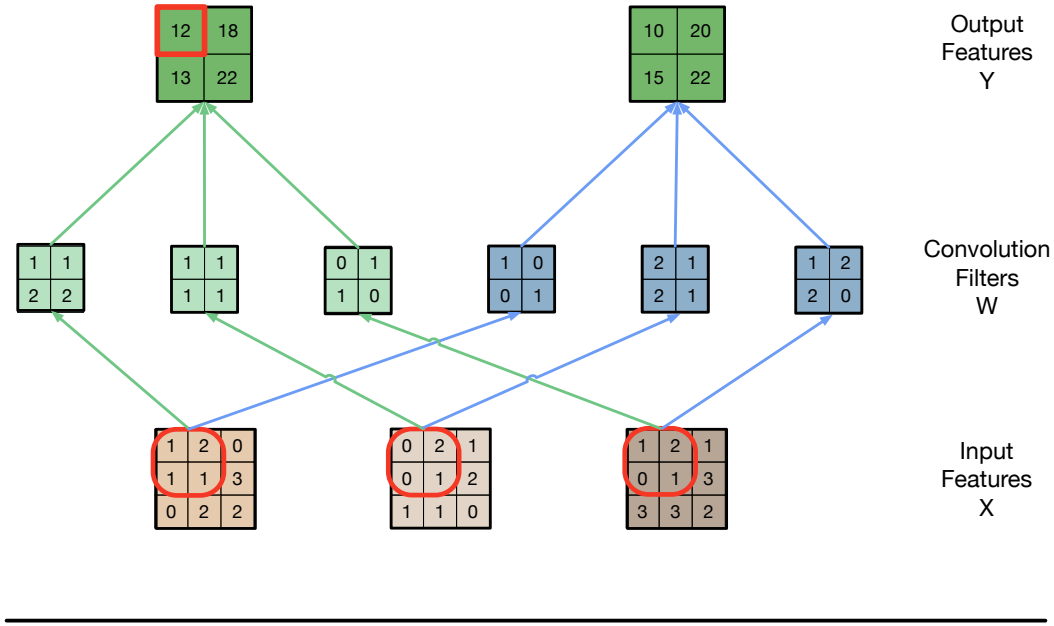
Some Observations (2)

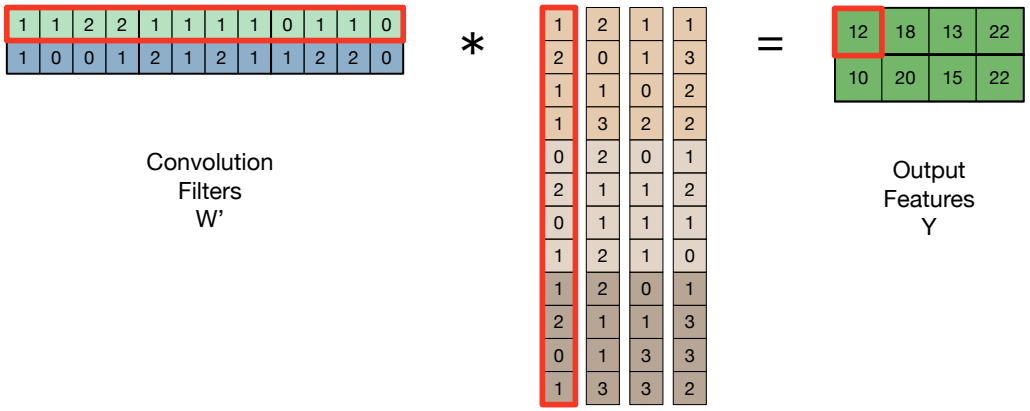
- Each input tile is loaded from global memory multiple times (by multiple blocks), once for each block that calculates the output tile that requires the input tile
  - Not very efficient in global memory bandwidth
  - This becomes more of a problem when the output feature maps become small (and increase in number)

Reducing Convolution Layers to Matrix Multiplications

- Convolution layers are the compute intensive parts of a CNN
- GPUs have extremely high-performance implementations of matrix multiplications
- Tiling techniques for matrix multiplication naturally reuse input features across output feature maps
- Converting convolutions in a convolution layer to a matrix multiplication helps to keep the level of parallelism stable across CNN layers

Implementing a Convolutional Layer with Matrix Multiplication



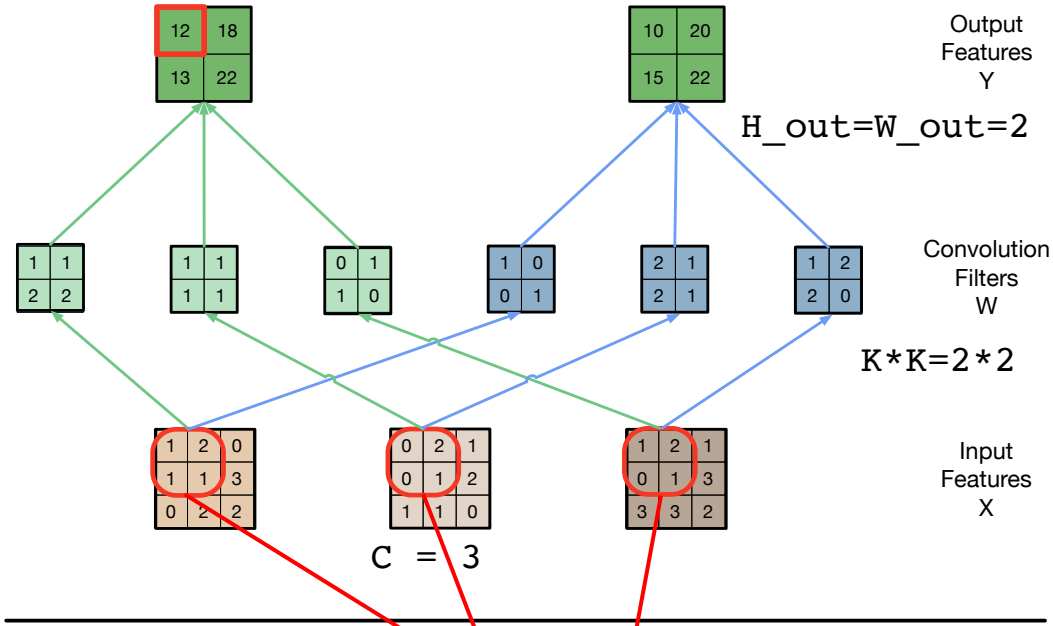


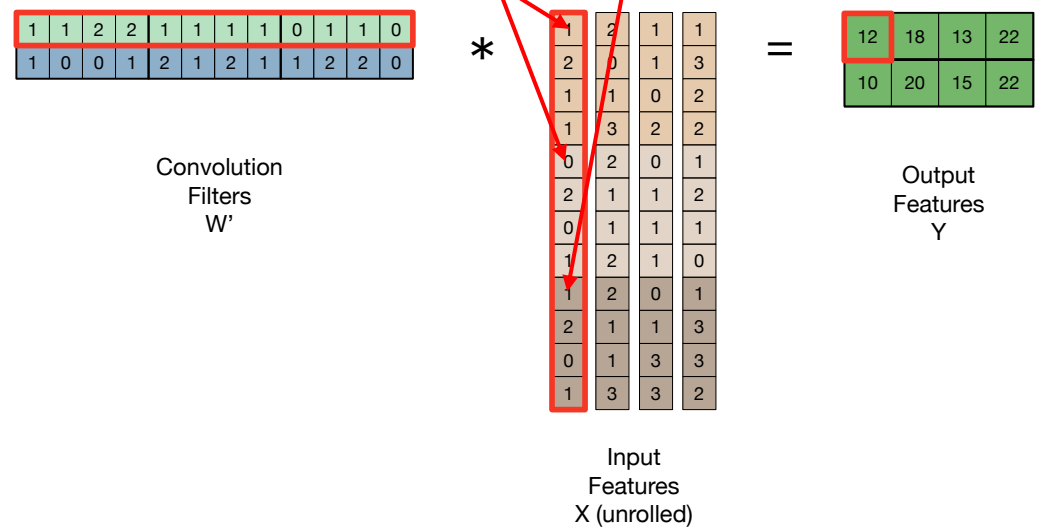
Slide credit: Reproduced from Hwu & Kirk

A C Function that Generates "Unrolled" X

```
01 void unroll(int C, int H, int W, int K, float* X, float* X_unroll) {
02   int H_out = H – K + 1;
03   int W_out = W – K + 1;
04   for(int c = 0; c < C; c++) {
     // beginning row index of the section for channel C input feature
     // map in the unrolled matrix
05     w_base = c * (K*K);
06     for(int p = 0; p < K; p++)
07       for(int q = 0; q < K; q++) {
08         int h_unroll = w_base + p*K + q;
09         for(h = 0; h <  H_out; h++)
10           for(int w = 0; w < W_out; w ++) {
11             int w_unroll = h * W_out + w;
12             X_unroll[h_unroll, w_unroll) = X(c, h + p, w + q);
13           }
14         }
15       }
16     }
17   }
18 }
```

Implementing a Convolutional Layer with Matrix Multiplication



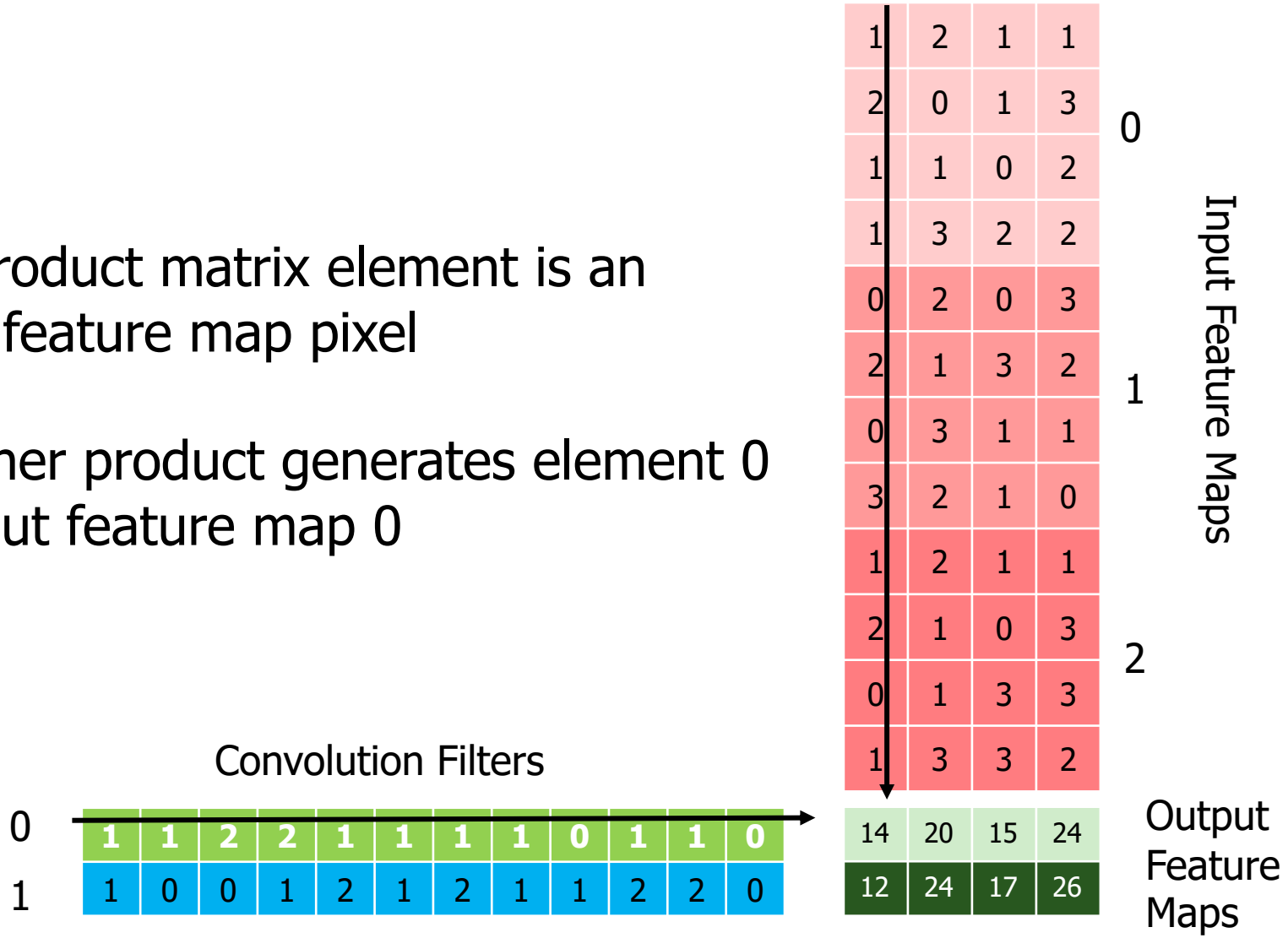



```
01 void unroll(int C, int H, int W, int K,
                          float* X, float* X_unroll)
{
02   int H_out = H – K + 1;
03   int W_out = W – K + 1;
04   for(int c = 0; c < C; c++) {
// beginning row index of the section for channel
// C input feature map in the unrolled matrix
05     w_base = c * (K*K);
06     for(int p = 0; p < K; p++)
07       for(int q = 0; q < K; q++) {
08         int h_unroll = w_base + p*K + q;
09         for(h = 0; h <  H_out; h++)
10           for(int w = 0; w < W_out; w ++) {
11             int w_unroll = h * W_out + w;
12             X_unroll[h_unroll, w_unroll) =
                          X(c, h + p, w + q);
13           }
14         }
15       }
16      }
17    }
18  }
```

Simple Matrix Multiplication

Each product matrix element is an output feature map pixel

This inner product generates element 0 of output feature map 0

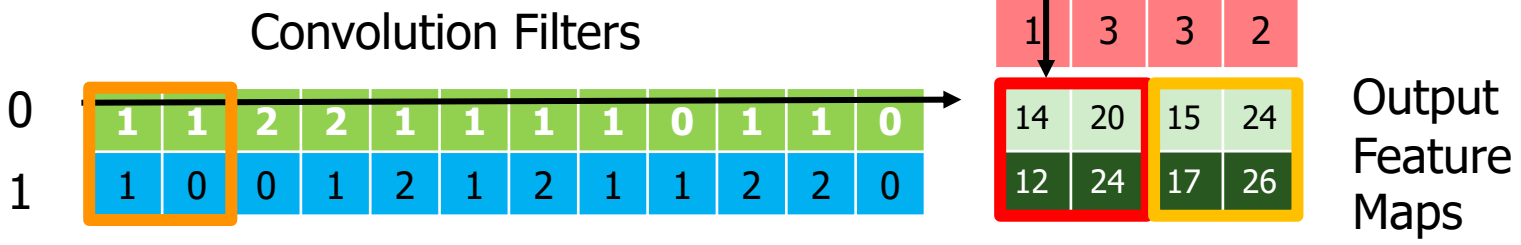


Tiled Matrix Multiplication 2x2 Shared-Memory Tiling (Toy Example)

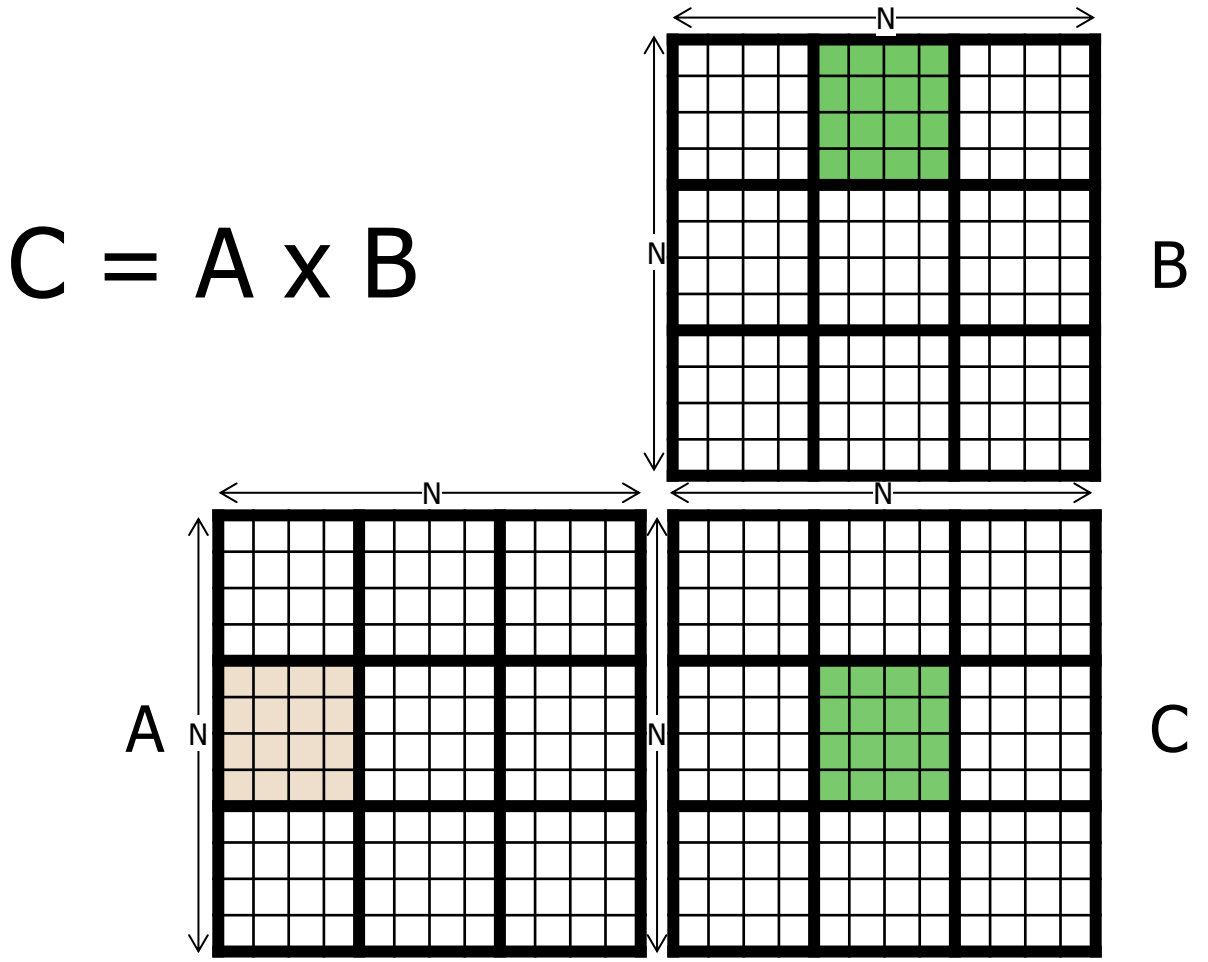
Each block calculates one 2x2 output tile – 2 elements from each output map

Each block loads one 2x2 filter tile and one 2x2 input tile into the shared memory and performs two dot product steps for all its 2x2 output elements

Each input element is reused 2 times in the shared memory



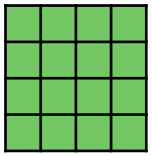
Tiled Matrix-Matrix Multiplication (I)



Step 1: Load the first tile of each input matrix to shared memory (each thread loads one element)

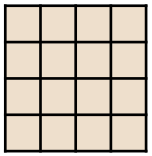
Slide credit: Izzat El Hajj

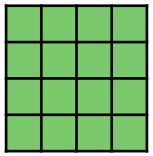
Tiled Matrix-Matrix Multiplication (II)



B_{tile}

$$C_{tile} = A_{tile} \times B_{tile}$$

A_{tile} 



C_{tile}

Step 2: Each thread computes its partial sum from the tiles in shared memory (threads wait for each other to finish)

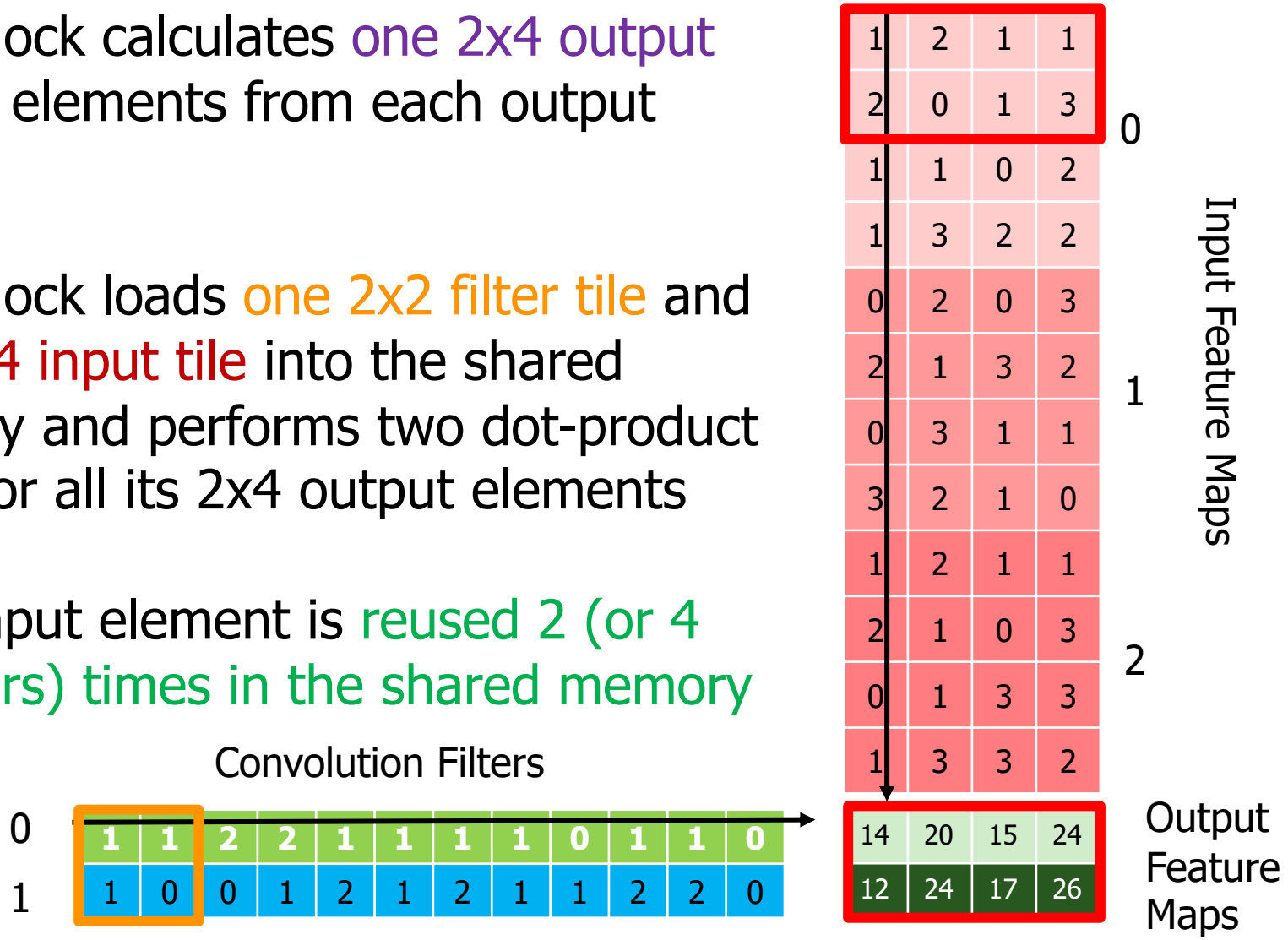
Slide credit: Izzat El Hajj

Tiled Matrix Multiplication 2x4 Shared-Memory Tiling (Another Toy Example)

Each block calculates one 2x4 output tile – 4 elements from each output map

Each block loads one 2x2 filter tile and one 2x4 input tile into the shared memory and performs two dot-product steps for all its 2x4 output elements

Each input element is reused 2 (or 4 for filters) times in the shared memory



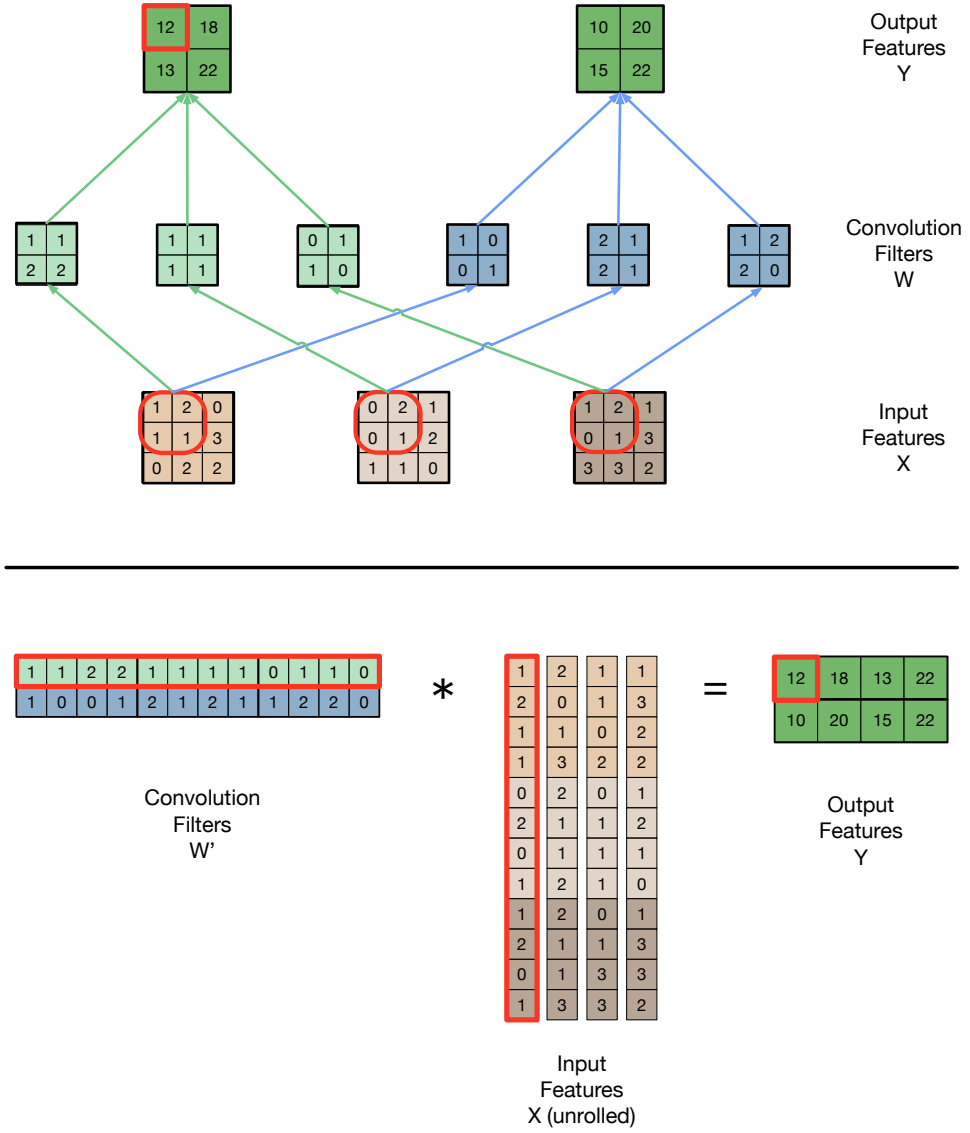
Analysis of Efficiency: Total Input Replication (I)

- Each output map element requires its own replicated K*K input feature map elements
  - Not replicated for different output feature maps
  - There are H_out * W_out output feature map elements
  - Each requires K * K replicated input feature map elements
  - So, the total number of input elements after replication is H_out * W_out * K * K times for each input feature map
  - The total number of elements in each original input feature map is (H_out + K - 1) * (W_out + K - 1)

$$\text{Expansion ratio} = \frac{C * H\_out * W\_out * K * K}{C * (H\_out + K - 1) * (W\_out + K - 1)}$$

Analysis of Efficiency: Total Input Replication (II)

- H_out = 2
- W_out = 2
- K = 2
- There are C=3 input maps (channels)
- The total number of input elements in the replicated ("unrolled") input matrix is 3*2*2*2*2
- The expansion ratio is (3*2*2*2*2)/(3*3*3) = 1.78



Tiled Matrix Multiplication is More Stable in Matrix Sizes

- The filter-bank matrix is an M * C * K * K matrix
  - M is the number of output feature maps
- The expanded input feature map matrix is a C * K * K * H_out * W_out matrix
- The sizes of the matrices depend on products of the parameters to the convolution, not the parameters themselves
- For example, while H_out * W_out tends to decrease towards later stages of a CNN, C tends to increase at the same time
  - The amount of work for each kernel launch will remain large as we go to the later stages of the CNN

More Details of Unrolling the Input Feature Maps (X)

- The conversion from convolution to matrix multiplication can be done during execution
  - The input feature maps are stored in their original form
  - The kernel that implements a convolution layer performs a tiled matrix multiplication on the conceptual unrolled input matrix
  - When loading each tile from the "unrolled input matrix", the kernel extracts the tile elements from the original input feature maps based on a mapping like that used in the C code
  - This way, there is no preprocessing cost and the input feature maps in the memory is not expanded due to unrolling

Advanced Tiling Techniques for Matrix Multiplication

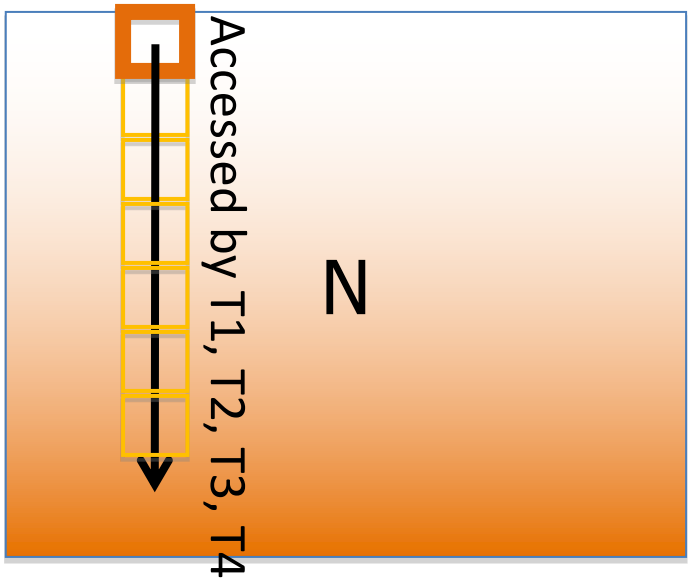
Joint Register and Shared-Memory Tiling

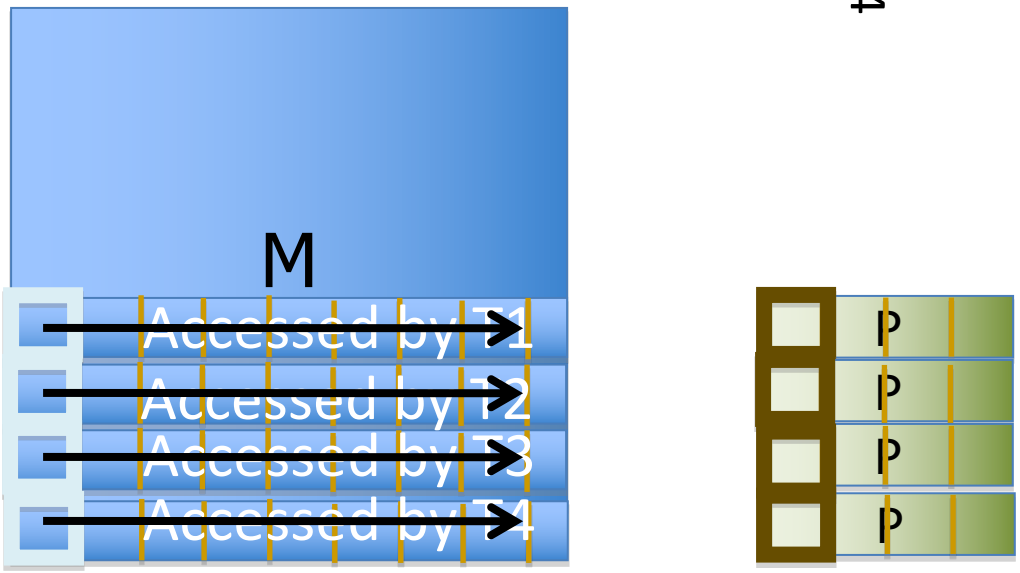
- Registers are accessed at extremely high throughput but
  - Private to each thread
  - Register tiling needs thread coarsening
- Shared memory is accessed at lower throughput than registers but still much higher than global memory
  - Visible to all threads in a block
  - Does not need thread coarsening
  - Still needs to be first loaded into registers before used by compute units
- We typically use both for tiling different dimensions of a multidimensional data
- One can use registers even more aggressively with warp programming and tensor cores

Slide credit: Hwu & Kirk & PUMPS+AI 2022

Data Reuse in Matrix Multiplication (Revisited)

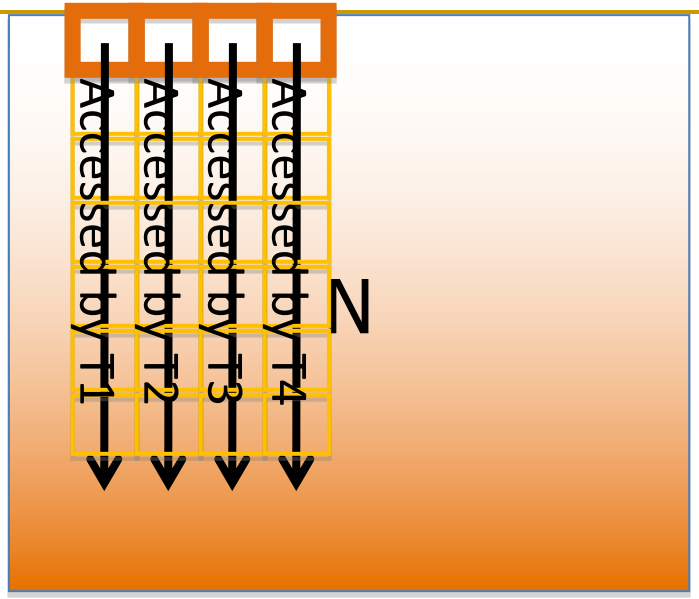
Reuse of column data in N

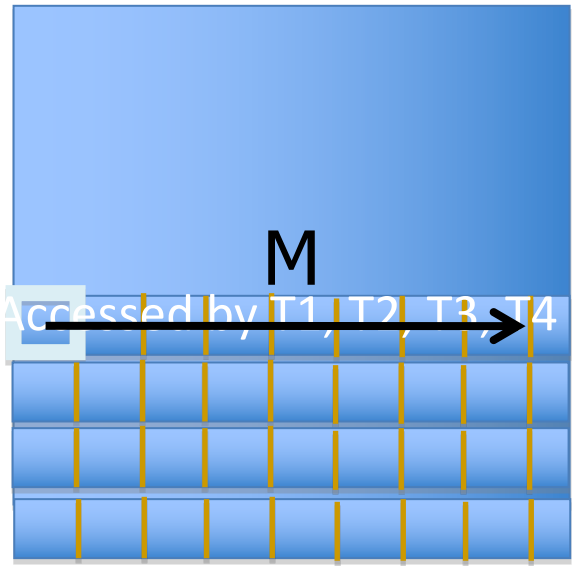


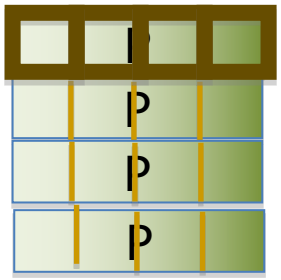


Data Reuse in Matrix Multiplication (Revisited)

Reuse of row data in M



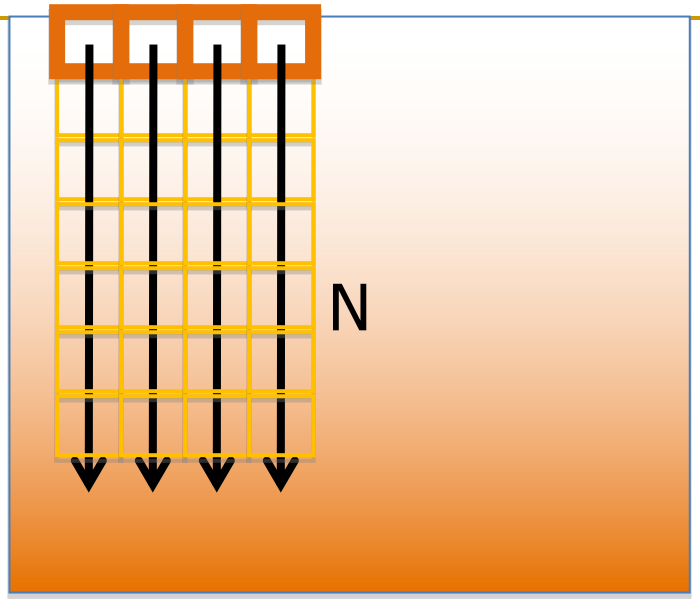


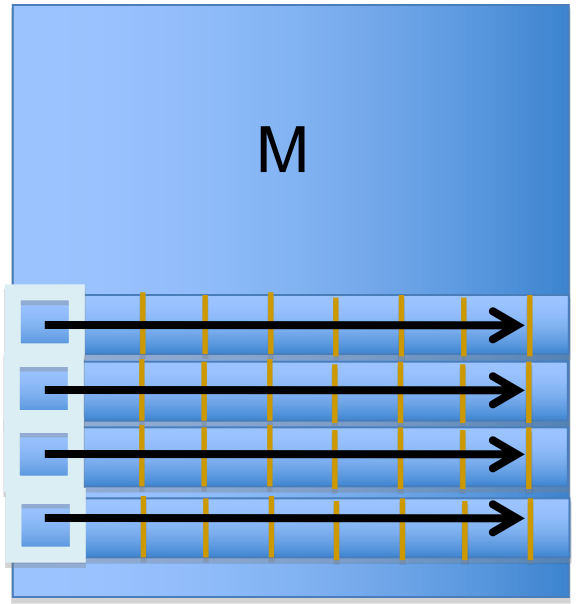


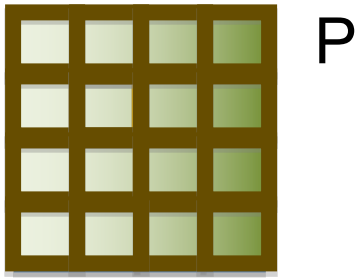
Slide credit: Hwu & Kirk & PUMPS+AI 2022

Data Reuse in Matrix Multiplication (Revisited)

- Only four elements of M and four elements of N are needed to calculate one step for a 16-element tile of P



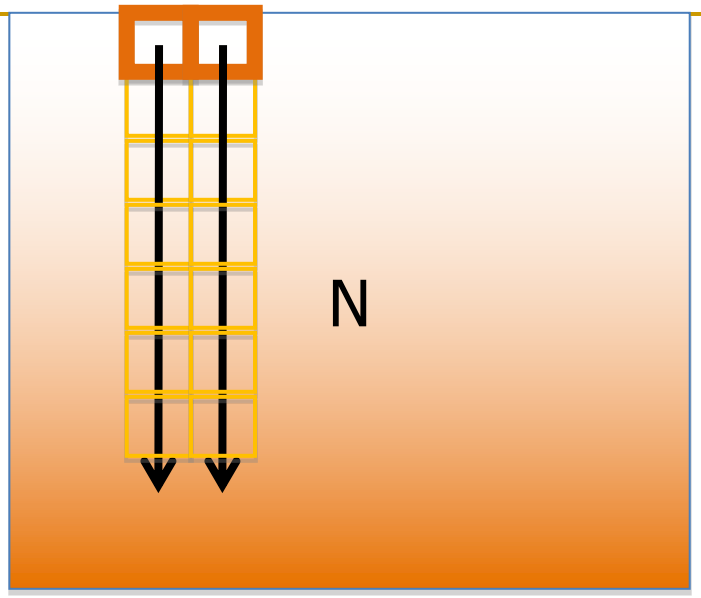


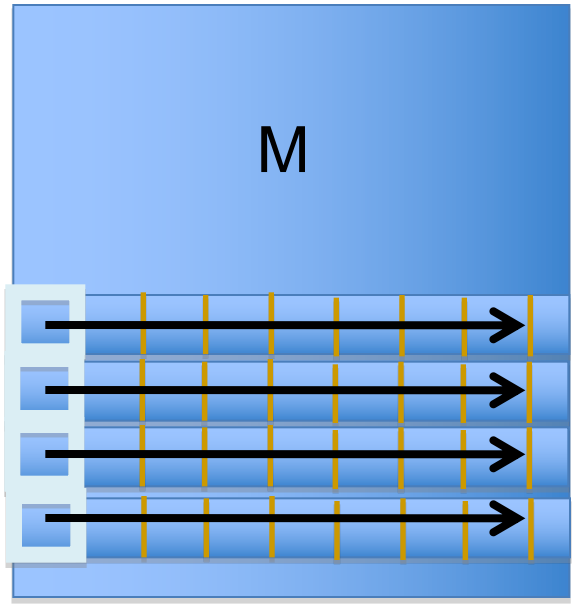


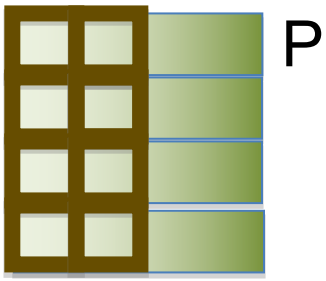
Slide credit: Hwu & Kirk & PUMPS+AI 2022

Data Reuse in Matrix Multiplication (Revisited)

- The P output tile does not need to be square
- For a 4x2 tile
  - 4 elements of M and 2 elements of N are needed for each step



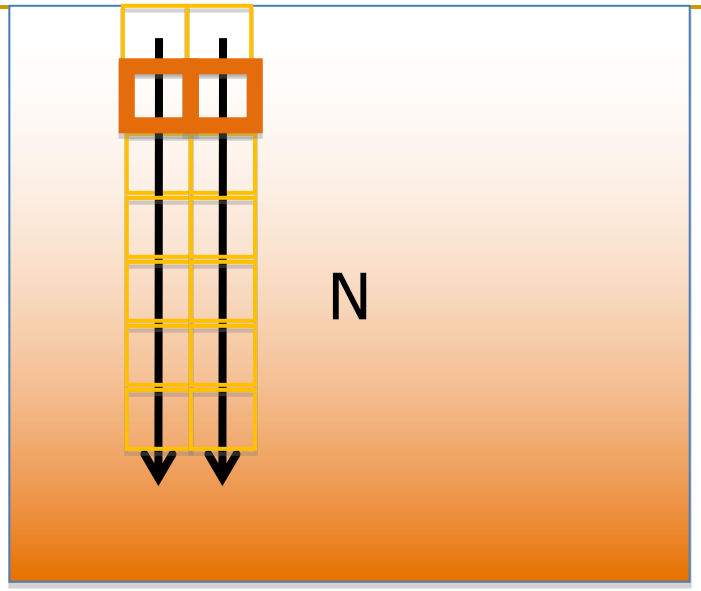


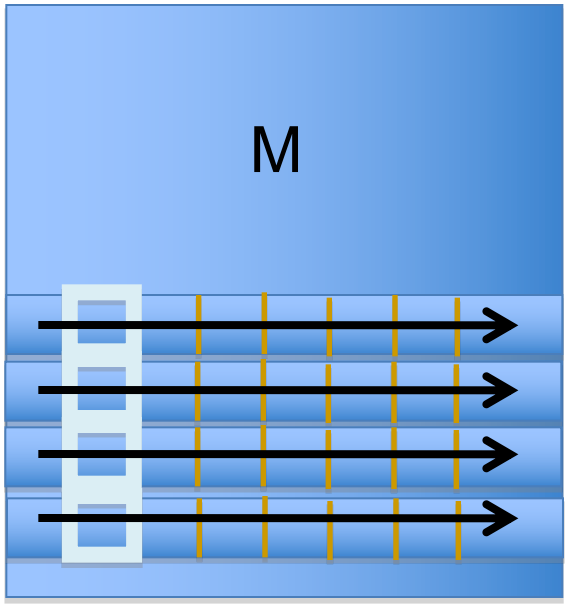


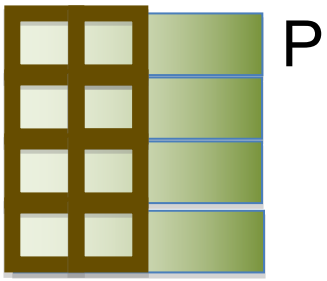
Slide credit: Hwu & Kirk & PUMPS+AI 2022

Data Reuse in Matrix Multiplication (Revisited)

- Step 2…



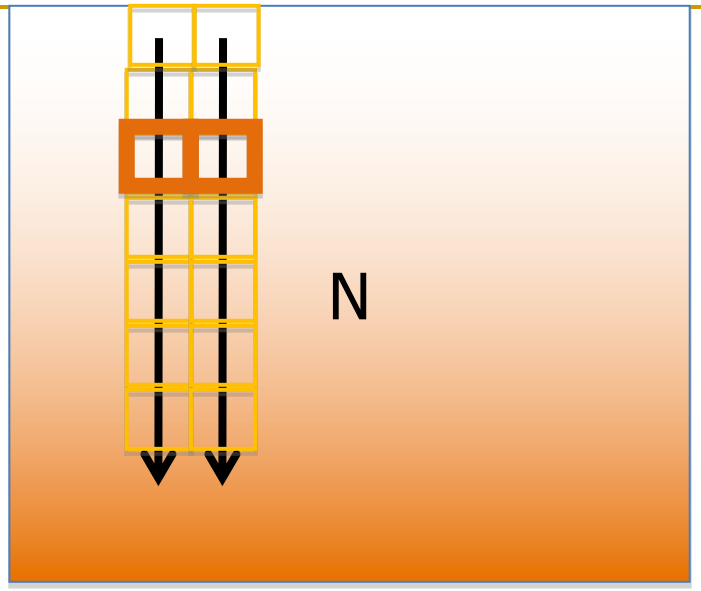


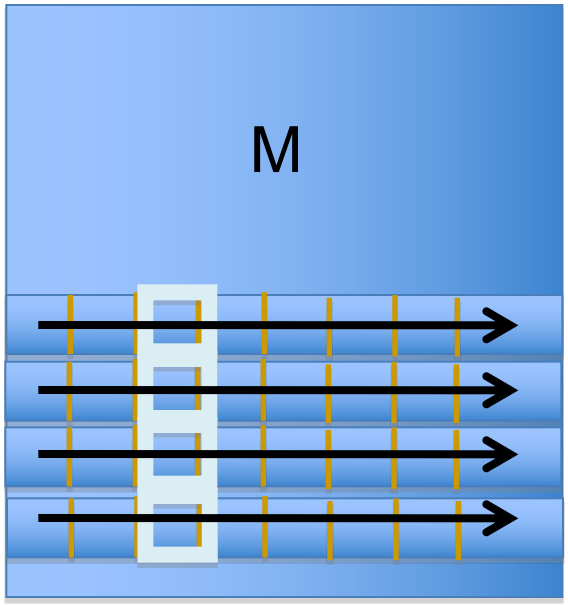


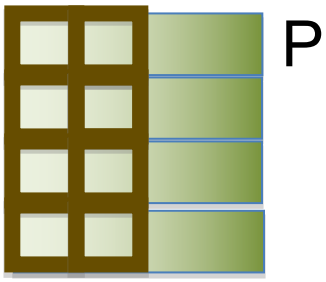
Slide credit: Hwu & Kirk & PUMPS+AI 2022

Data Reuse in Matrix Multiplication (Revisited)

- At each step
  - For 4x2, only 6 elements need to be loaded for all 8 threads to make progress
  - For 4x4, 8 elements for all 16 threads







Slide credit: Hwu & Kirk & PUMPS+AI 2022

In the Kernel of the Previous Slide (e.g., 4x4 Output Tile)

- TILE_WIDTH^2 elements of M and TILE_WIDTH^2 element of N are loaded
- Each thread block calculates TILE_WIDTH steps for TILE_WIDTH^2 elements of P
  - TILE_WIDTH is typically at least 16
- According to our analysis, we can use much smaller amount of shared memory by
  - Loading TILE_WIDTH element of M and TILE_WIDTH element of N (input tiles) to calculate 1 step for TILE_WIDTH^2 elements in the output tile
  - So, why didn't we do so?

Cost of Loading Smaller Input Tile

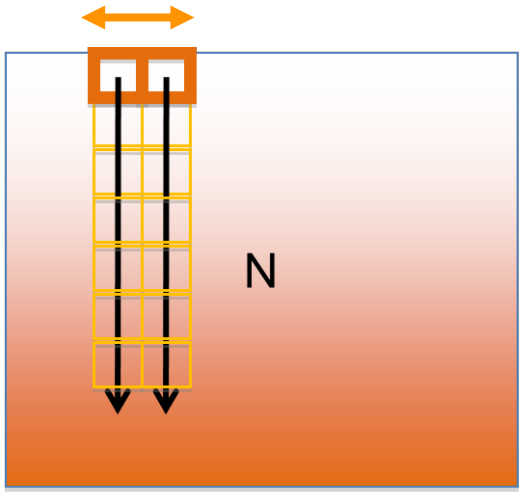
- If in each iteration of the outer loop
  - only a subset of threads load M and N elements (divergence or load imbalance)
  - Call __synchthreads()
  - All threads calculate one step of the inner product (inner loop degenerates to one iteration)
  - Call __synchthreads()
  - Go to the next iteration
- Even though __synchthreads() is a very efficient function, such intensive use is still going to hurt

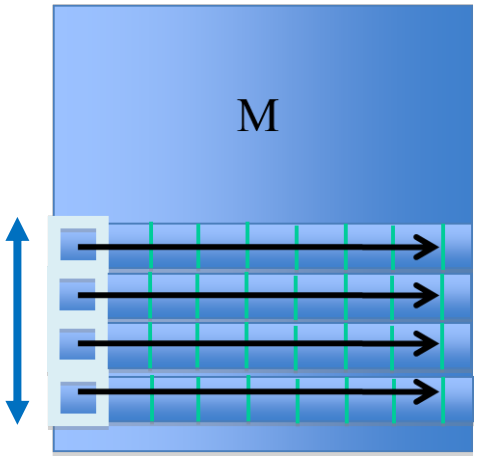
Joint Register and Shared Memory Tiling

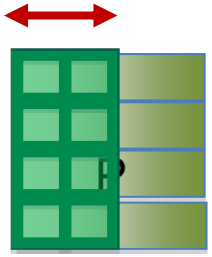
- Store input M tile and output P tile elements in registers
- Store input N tile elements in shared memory
- Decouple of M and N input tile widths
  - TILE_WIDTH_M , TILE_WIDTH_N
- Key quantities
  - Number of threads = TILE_WIDTH_M
  - Output tile size = TILE_WIDTH_M * TILE_WIDTH_N
  - Reuses for each N element = TILE_WIDTH_M
  - Reuses for each M element = TILE_WIDTH_N
  - Each thread calculates TILE_WIDTH_N P elements

Example:

TILE_WIDTH_M = 4
TILE_WIDTH_N = 2

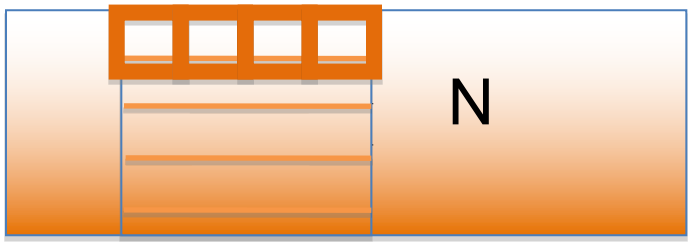


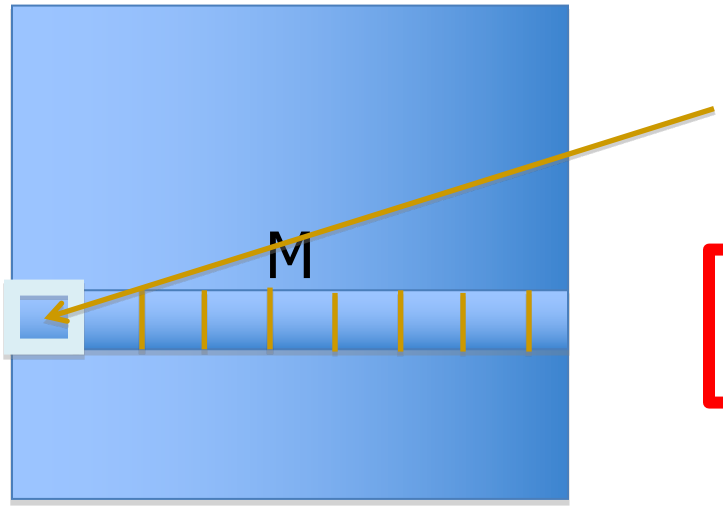




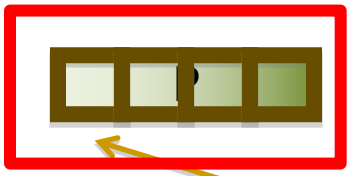
Joint Register and Shared Memory Tiling Optimization 1: Thread Coarsening

- Have each thread to calculate a horizontal strip of TILE_WIDTH_N P elements
- Data loaded from M can be reused TILE_WIDTH_N times through registers
  - Register tiling





Load 1 value from M into r1

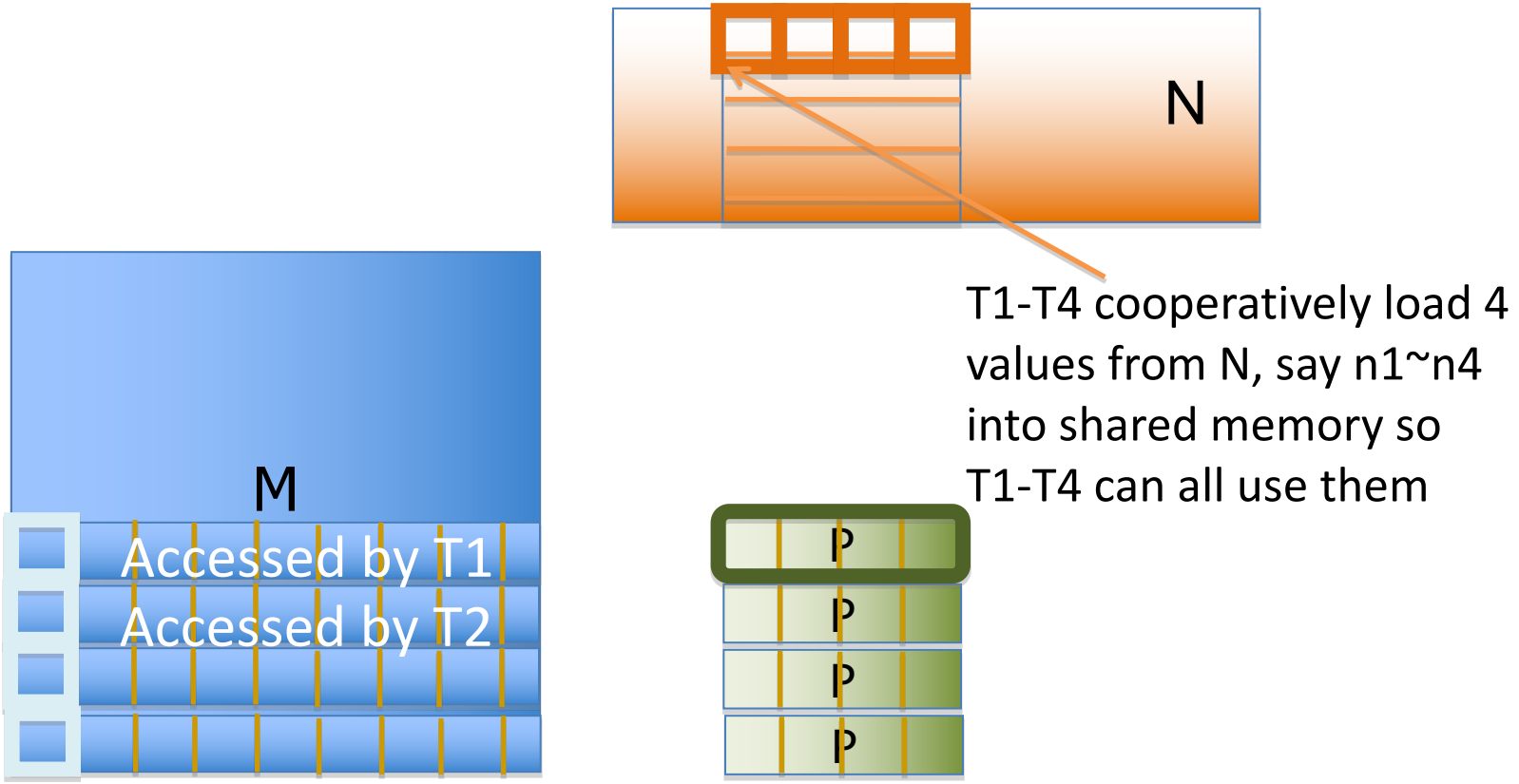


r1 can be reused to compute 4 intermediate results for P

```
P[0] += r1 * n1;
P[1] += r1 * n2;
P[2] += r1 * n3;
P[3] += r1 * n4;
```

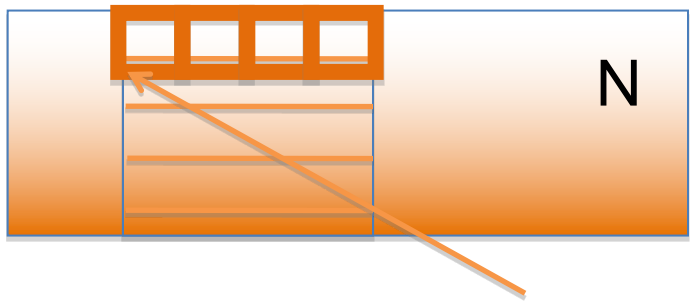
Joint Register and Shared Memory Tiling
Optimization 2: Shared Memory Tiling

- Multiple threads collaborate to load TILE_WIDTH_N N elements into shared memory

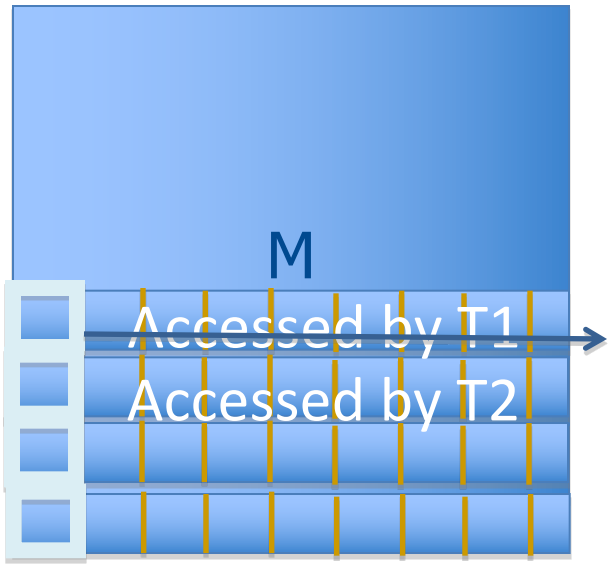


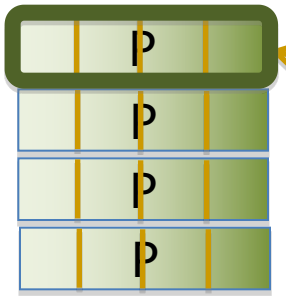
In One Iteration, Each Thread…

- Accesses one M element from register, accesses TILE_WIDTH_N N elements from shared memory
  - Calculates one step for TILE_WIDTH_N P elements
  - TILE_WIDTH_N = ~16 in practice (limited by registers needed for P elements)



n1~n4 accessed from shared memory





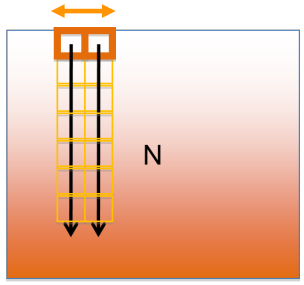
Intermediate results computed by T1; stored in registers

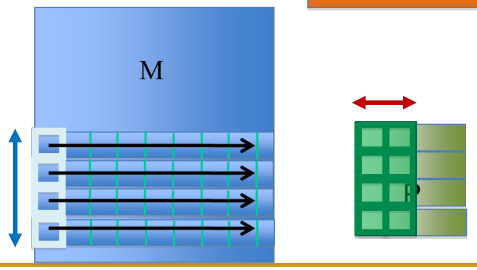
In One Iteration, Each Block…

- Has TILE_WIDTH_M threads
  - 64 or more in practice

- Loads TILE_WIDTH_M M elements into registers, loads TILE_WIDTH_N N elements into shared memory
  - TILE_WIDTH_N is number of threads folded into one thread in thread coarsening (16 or more in practice)

- However, loading of N will involve only a subset of threads (divergence)

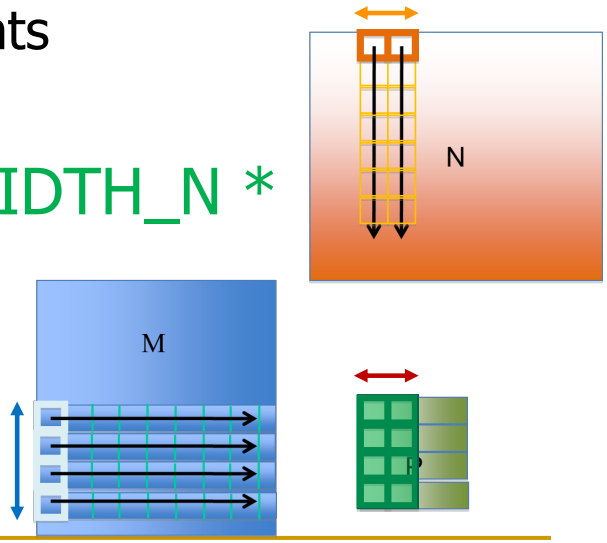




Slide credit: Hwu & Kirk & PUMPS+AI 2022

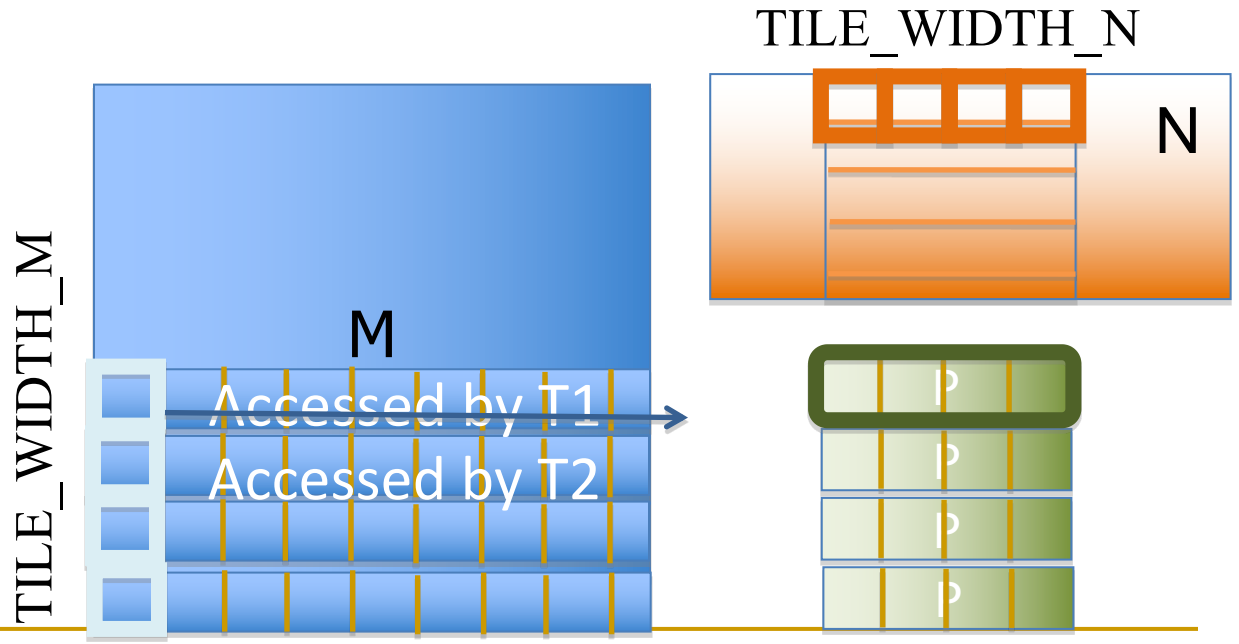
A More Balanced Approach

- In each iteration, all threads in a block collaborate to load a TILE_WIDTH_N * K tile of N into shared memory
  - K is set so that TILE_WIDTH_M = TILE_WIDTH_N * K
  - Every thread loads one N element, no divergence
- Each thread loads K M elements into registers
  - Value of K is limited by the number of registers
  - Each thread needs to use
    - TILE_WIDTH_N registers for output elements
    - K registers for M elements
- Each block calculates K steps for TILE_WIDTH_N * TILE_WIDTH_M P elements



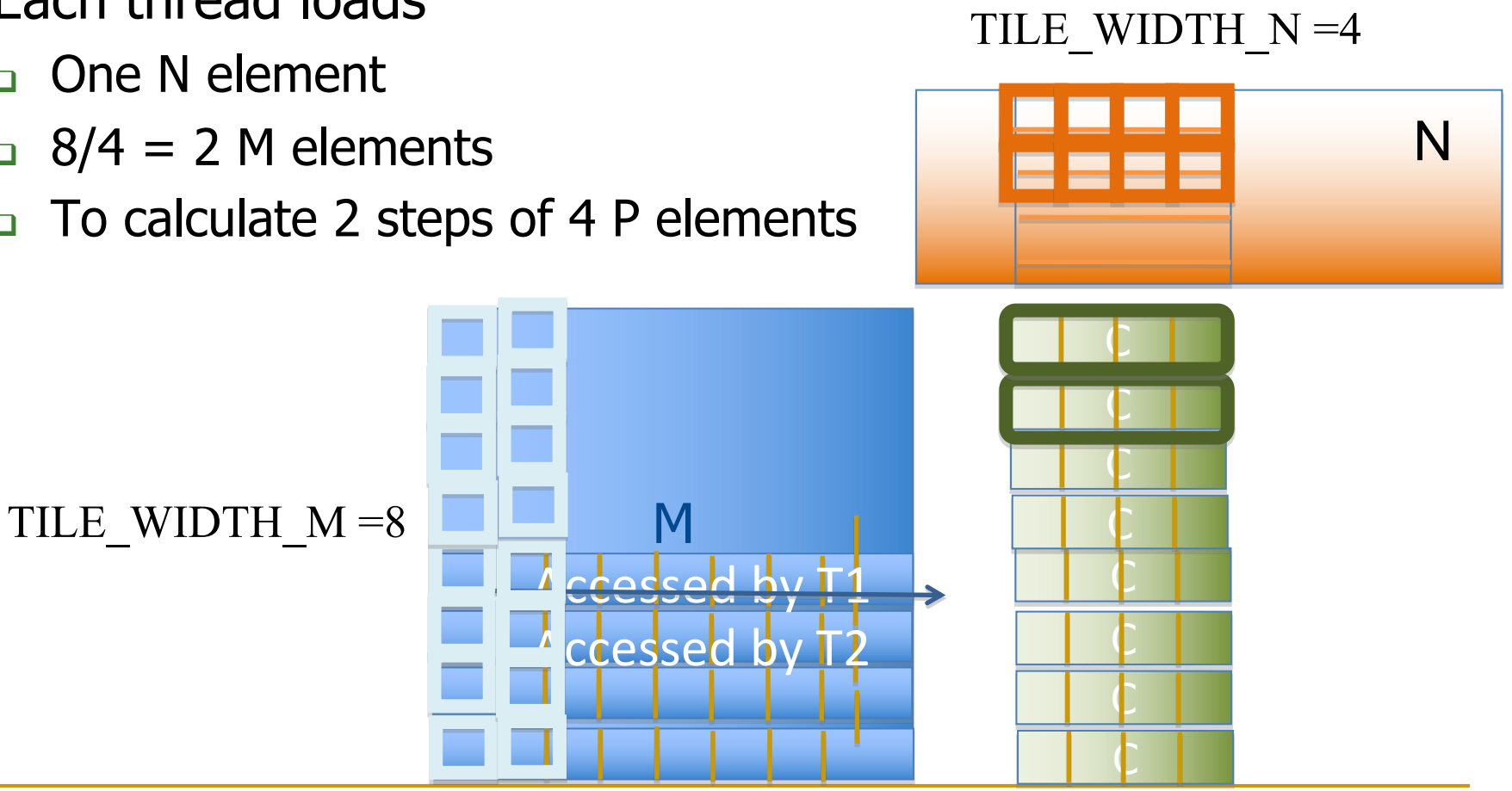
Summary: Joint Register and Shared Memory Tiling

- Each block has TILE_WIDTH_M threads
- Each thread coarsened by TILE_WIDTH_N times
- Each thread loads
  - One N element into the shared memory
  - K = TILE_WIDTH_M/TILE_WIDTH_N M elements
  - To calculate K steps of TILE_WIDTH_N P elements



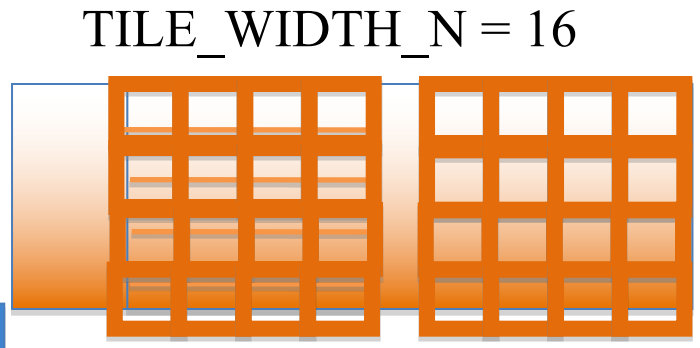
For a Toy Example

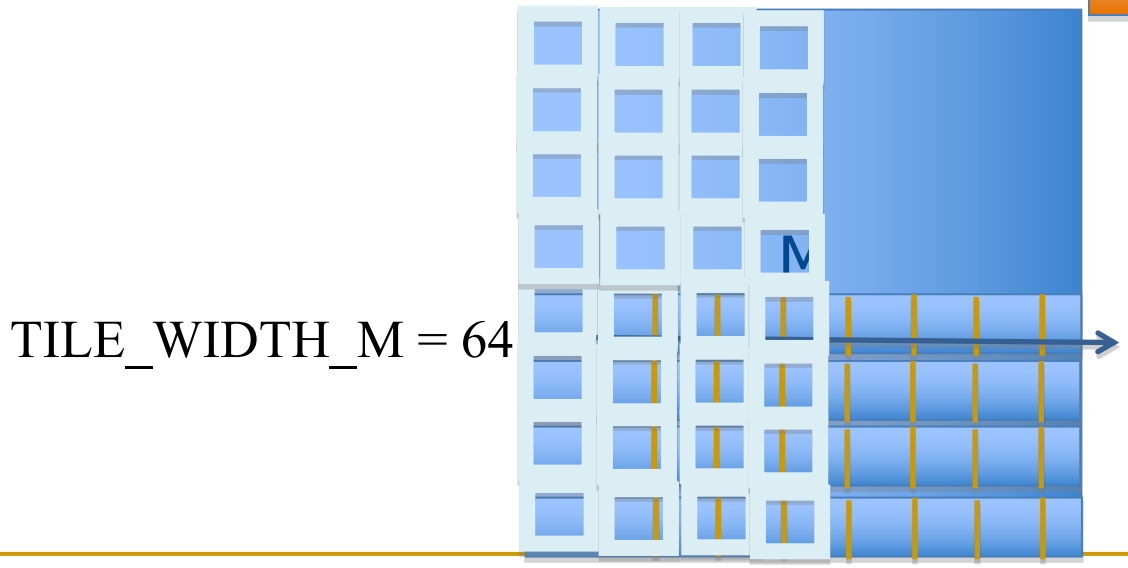
- Each block has 8 threads
- Each thread coarsened by 4 times
- Each thread loads
  - One N element
  - 8/4 = 2 M elements
  - To calculate 2 steps of 4 P elements



For GTX280 (Volkov & Demmel)

- Each block has 64 threads
- Each thread coarsened by 16 times
- Each thread loads
  - One N element
  - 64/16 = 4 M elements
  - To calculate 4 steps of 16 P elements





Slide credit: Hwu & Kirk & PUMPS+AI 2022

A Comparative Analysis

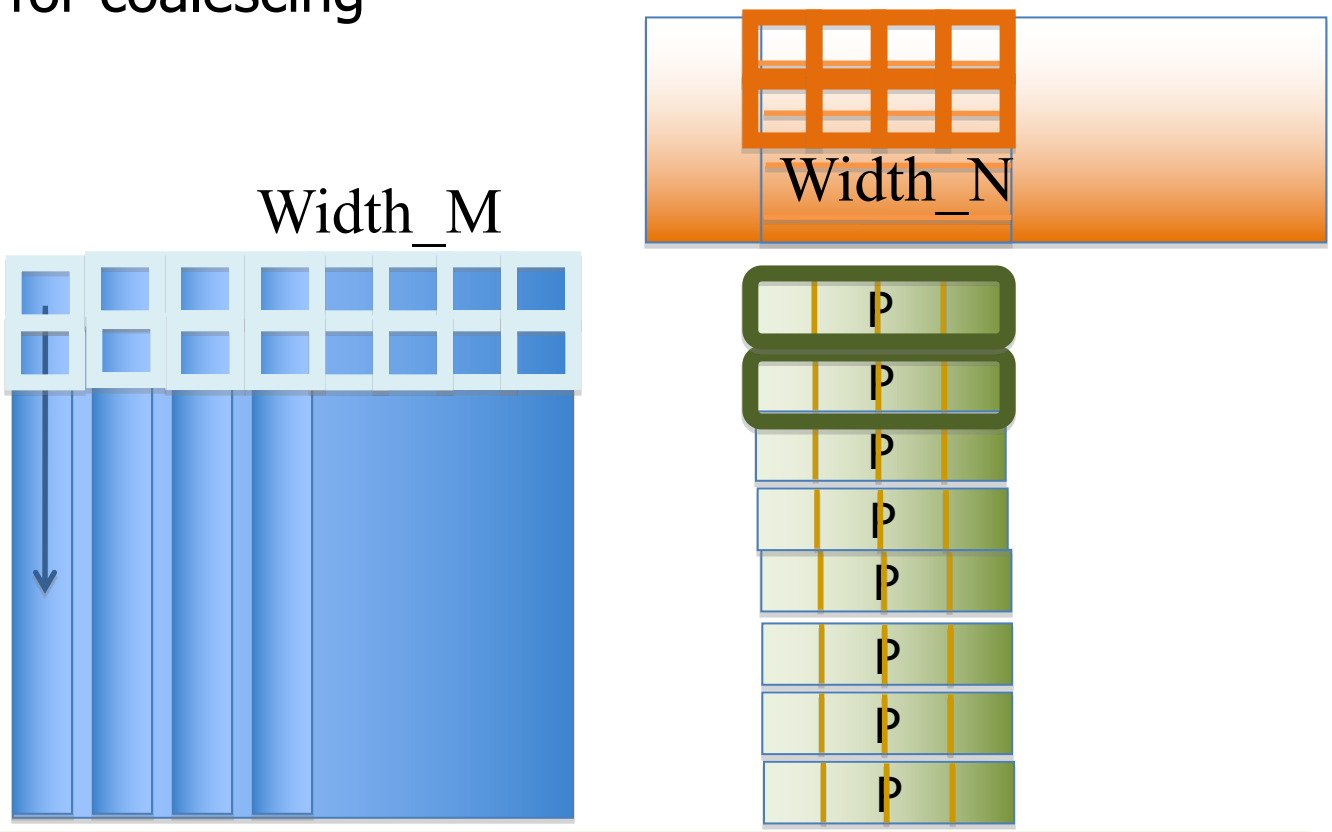
- Tiled SGEMM using shared memory for both inputs:
  - Each thread block computes 32x32=1024 results
  - Uses 12 KB on-chip memory (register + shared memory)
- Register/Shared-Memory tiled version of SGEMM:
  - Each thread block computes 64x16=1024 results
  - Uses only 5 ¼ KB on-chip memory
    - Similar degree of reuse; ~2X more efficient

Tiling algorithm	# of reuse per data in M	# of reuse per data in N	# of data computed per block in P	Shared memory usage per block	(M+P) Register usage per TB	Performance on GTX280 in GFLOP/s
Jointly tiled SGEMM	16	64	16x64	(N) 4x16x4 =256Bytes	(64x4+64x16)x4 =5KB	~430
Shared-Memory Tiled SGEMM	32	32	32x32	(N+M) 32x32x4x2 =8KBytes	32x32x4=4KB	<300

Slide credit: Hwu & Kirk & PUMPS+AI 2022

Data Layout – For C (Row Major)

- Loading N into shared memory is easily coalesced with the 16x4 tile
- Loading M into registers in not coalesced
  - Transpose M for coalescing



CUDNN Library

CUDNN Library

- CUDNN is a library of optimized routines for implementing deep learning primitives
- C-language API that integrates into existing deep learning frameworks (e.g., Caffe, Tensorflow, Theano, Torch)
- Same as cuBLAS, CUDNN assumes that input and output data reside in the GPU device memory

Slide credit: Hwu & Kirk & PUMPS+AI 2022

Batched Convolution in CUDNN (I)

- Most important primitive in CNN

- Two inputs to the convolution:
  - D, a four-dimensional N x C x H x W tensor, with input data
  - F, a four-dimensional K x C x R x S tensor, with convolutional filters

Parameter	Meaning
N	Number of images in minibatch
C	Number of input feature maps
H	Height of input image
W	Width of input image
K	Number of output feature maps
R	Height of filter
S	Width of filter
u	Vertical stride
v	Horizontal stride
pad_h	Height of zero padding
pad_w	Width of zero padding

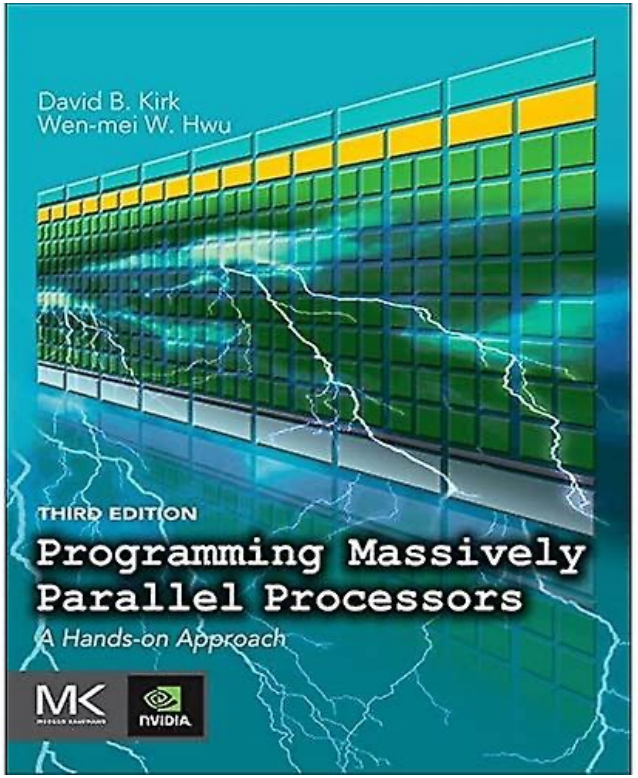
Batched Convolution in CUDNN (II)

- The output is also a four-dimensional tensor O
  - N x K x P x Q
  - P = f(H, R, u, pad_h)
  - Q = f(W, S, v, pad_w)
  - Height and width of the output feature maps depend on the input feature map and filter bank height and width, along with padding and striding choices
    - The goal of the striding parameters (u, v) is to reduce the computational load by computing only a subset of the output pixels
    - Padding parameters are for improved memory alignment and/or vectorized execution

Slide credit: Hwu & Kirk & PUMPS+AI 2022

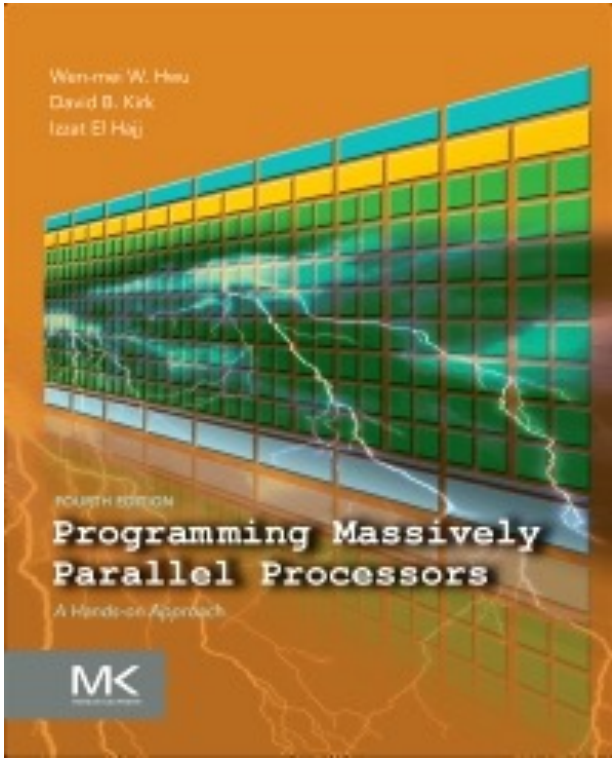
Recommended Readings (I)

- Hwu and Kirk, "Programming Massively Parallel Processors," Third Edition, 2017
  - Chapter 7 - Parallel patterns — convolution: An introduction to stencil computation
  - Chapter 16 - Application case study — machine learning



Recommended Readings (II)

- Hwu and Kirk and El Hajj, "Programming Massively Parallel Processors," Fourth Edition, 2022
  - Chapter 7 - Convolution: An introduction to constant memory and caching
  - Chapter 8 - Stencil
  - Chapter 16 - Deep learning



P&S Heterogeneous Systems

Advanced Tiling for Matrix Multiplication

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich

Fall 2022

28 November 2022